2

Technical Report

CMU/SEI-93-TR-05
ESC-TR-93-182

Carnegie-Mellon University
Software Engineering Institute

Safety-Critical Software: Status Report and Annotated Bibliography

Patrick R. H. Place
Kyo C. Kang

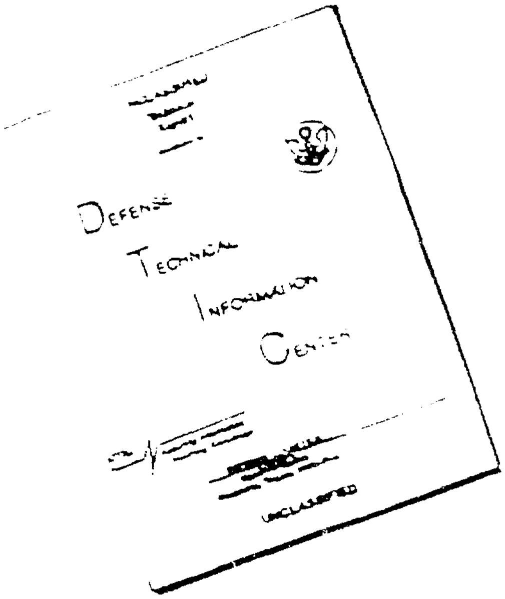
June 1993

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment or administration of its programs on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state or local laws, or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state or local laws, or executive orders. While the federal government does continue to exclude gays, lesbians and bisexuals from receiving ROTC scholarships or serving in the military, ROTC classes on this campus are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pa. 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pa. 15213, telephone (412) 268-2056.

DISCLAIMER NOTICE



DEFENSE
TECHNICAL
INFORMATION
CENTER

UNCLASSIFIED

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Technical Report
CMU/SEI-93-TR-5
ESC-TR-93-182
June 1993

Safety-Critical Software: Status Report and Annotated Bibliography

Patrick R. H. Place
Kyo C. Kang

Requirements Engineering Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/ENS
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.

This report was funded by the U.S. Department of Defense.

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Copies of this document are also available from Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212, Telephone (412) 321-2992 or 1-800-685-6510, Fax (412) 321-2994

Table of Contents

Safety-Critical Software: Status Report and Annotated Bibliography

Abstract: Many systems are deemed safety-critical and these systems are increasingly dependent on software. Much has been written in the literature with respect to system and software safety. This report summarizes some of that literature and outlines the development of safety critical software. Techniques for hazard identification and analysis are discussed. Further, techniques for the development of safety-critical software are mentioned. A partly annotated bibliography of literature concludes the report.

1 Introduction

This chapter discusses the reasons for writing this report and the role of safety-critical software in requirements engineering. Some background material suggesting reasons for the current increase in interest in safety-critical software is presented.

1.1 Purpose of This Report

The purpose of the report is to bring together concepts necessary for the development of software in safety-critical systems. An annotated bibliography may be used as a reference base for further study.

Although this report was produced by members of the requirements engineering project it covers aspects of software development outside the restricted area of requirements engineering. This is due, in part, to the nature of the literature surveyed, which discusses all aspects of software development for software in safety-critical systems. Also, the project members take the view that specification and analysis are part of the requirements engineering process and are activities performed as soon as system requirements have been elicited from the appropriate sources.

The report is not intended as a tutorial on any specific technique, though some techniques are highlighted and discussed briefly. Interested readers should turn to appropriate literature for more detailed information on the use of the techniques described herein. There has been a great deal of recent activity in the application of formal methods to safety-critical software development and we will outline, later in this report, the classes of formal method and how they may be used. We do not concentrate on specific methods since a method should be chosen to match the system under construction. Instead, we discuss options where the developers may choose one type of method over another.

1.2 Requirements Engineering and Safety

Standards exist that state that all safety-critical components of a system must be developed in a particular way. Given that the required development techniques may be more costly than current techniques, or be within the capabilities of a limited number of the staff, it is important to minimize the proportion of the system that has to be developed according to the safety standard. The requirements engineer has the opportunity to manipulate the requirements to minimize the safety-critical subsystems while maintaining an overall required level of safety for the entire system. Generally, a well-designed system will have few safety-critical components in proportion to the total system. However, these components may prove to be some of the hardest components to develop since their design and development requires a system-level rather than a component-level understanding.

It is clear that safety must be considered from the start in the development of a system. This means considering issues of safety at the concept exploration phase, the demonstration and validation phase, and the full scale development phase. Safety concerns often conflict with other development concerns such as performance or cost. Decisions should not be made during development for reasons of performance or cost that compromise safety without performing an analysis of the risk associated with the resultant system. The safety of a system is considered by understanding the potential hazards of the system, that is, the potential accidents that the system may cause. Once the hazards are understood, the system may be analyzed in terms of the safety hazards of the components of the system, and each component may be analyzed in the same way, leading to a hierarchy of safety specifications.

The development of the requirements specification is a part of the requirements engineering phase; indeed, the product of requirements engineering should be the specification for use by the developers. During the requirements engineering phase, design decisions are made concerning the allocation of function to system components; it is at this stage that decisions concerning overall system safety must be made. The specification acts as the basis for both development and testing.

An important objective of requirements engineering is the elimination of errors in the requirements. These errors typically occur in two forms: misunderstanding customer desires or poorly conceived customer requests. The implication of this is that the requirements engineering process must analyze the requirements for both desirable and undesirable behaviors.

Safety is a system-level issue and cannot be determined by examining the safety of the components in isolation. The approach taken is to develop a system model which represents a safe system; if not, the system will never be safe since the model is used as the basis for analysis and further development. The developers are led into developing components of the system in isolation and the system integrators put these components together. Although each of the individual components may be safe, the integrated system may not be safe and may well be untestable for safety given the infeasibility of generating sufficient test cases for a reliable and safe system.

Many systems cannot be feasibly tested in a live situation. For example, systems such as nuclear power plant shutdown systems, aircraft flight control systems, or critical components of strategic weapons systems cannot be adequately tested because it would be necessary to create a hazardous situation in which failure would be disastrous.

Customers' requirements are usually presented in many forms, examples being natural language descriptions, engineering diagrams and mathematics. In order to engineer a safe system, it is generally the case that each customer's requirements be organized into a coherent form that may be analyzed in a cost-effective manner.

Formal specification techniques provide notations appropriate for the specification and analysis of systems or software that cannot be tested in a live situation. These techniques provide notations that may be used to model the customer's desires [186]. Instead of relying on potentially ambiguous natural language statements, the specifications describe the system using mathematics with only one possible interpretation, which may be analyzed for defects. When completed, the formal specification forms a model of the system and may be used to predict the behavior of that system under any given set of circumstances. Thus, the safety of the system may be estimated by using the model to predict how the system will react to a given sequence of potentially hazardous events. If the model behaves according to the customer's notions of safety, then we can have confidence that a system conforming to the specifications will be safe.

1.3 Background

The use of software is increasing in safety-critical components of systems being developed and delivered. Examples of systems using software in place of hardware in safety-critical systems are the Therac 25 (a therapeutic linear accelerator) and nuclear reactor shutdown systems (Darlington, Ontario, is the best publicized example). There are many other instances of introduction of software into safety-critical systems.

In many cases the new software components are replacing existing hardware components. The introduction of software into such systems introduces new modes of failure for the systems which cannot be analyzed by the traditional engineering techniques. This is because software fails differently from hardware; software failure is less predictable than hardware failure.

1.4 Structure of the Report

The report collects a number of topics relating to requirements engineering and the subsequent development of systems with safety-critical components. Chapter 2 is a collection of themes that recur throughout the literature with some commentary on each theme. Chapter 3 describes the various techniques used to determine which parts of a system are safety critical and which are not. Chapter 4 discusses development techniques applicable to the development of safety-critical systems throughout major phases of implementation. Chapter 5 describes a number of current standards pertaining to the development of software for safety-critical systems. This chapter also discusses some concerns on the usefulness of standards and some harmful effects that a standard may create. Chapter 6 discusses the conclusions drawn while writing this report. This chapter also discusses potential avenues for further work in requirements engineering and developrnent of software in safety-critical systems. The report concludes with an annotated bibliography of papers and books relating to safety-critical systems.

2 Comments on Software Safety

This chapter collects a number of the concepts relating to safety-critical software that may be found in various journals and books: an extensive bibliography may be found at the end of this report. Each section presents a different concept and some discussion of that concept.

2.1 Safety Is a System Issue

Leveson [117] and others make the point that safety is not a software issue; rather, it is a system issue. By itself, software does nothing unsafe. It is the control of systems with hazardous components or the providing of information to people who make decisions that have potentially hazardous consequences that leads to hazardous systems. Thus, software can be considered unsafe only in the context of a particular system.

At the system level, software may be treated as one or more components whose failure may lead to a hazardous system condition. Such a condition may result in the occurrence of an accident.

2.2 Safety Is Measured as Risk

Safety is an abstract concept. We inherently understand what we mean when we say, "This system is safe." Essentially, we mean that it will not cause harm either to people or property. However, this notion is too simple to be useful as a statement of safety. There are many systems that can be made completely safe, but making systems that safe may interfere with their ability to perform their intended function. An example would be a nuclear reactor—the system is perfectly safe, so long as no nuclear material is introduced into the system. Such a system is, of course, not useful. Thus, the definition of safety becomes related to *risk*. Risk may be defined as

$$Risk = \sum_{hazard} \mathcal{E}(hazard) \times P(hazard)$$

where $\mathcal{E}$*(hazard)* is a measure of the effects that may be caused by a particular mishap and *P(hazard)* is the probability that the mishap will occur.

We will not further define how risk may be measured. Examples of appropriate measures would be in terms of either human life or replacement or litigation costs. There are many other measures that may be chosen to assess risk. However, the point we must accept is that no system will be wholly safe. Instead, we must attempt to minimize the risk by either containing the hazard or reducing the probability that the hazard will occur.

2.3 Reliability Is Not Safety

It is important to distinguish between the terms *reliability* and *safety*. According to definitions from Deutsch and Willis [55], reliability is a measure of the rate of failure in the system that renders the system unusable, and safety is a measure of the absence of unsafe software conditions. Thus, reliability encompasses issues such as the system's correctness with regard to its specification (assuming a specification that describes a usable system) and the ability of the system to tolerate faults in components of or inputs to the system (whether these faults are transient or permanent). Safety is described in terms of the absence of hazardous behaviors in the system.

As can be seen, reliability and safety are different system concepts: the former describes how well the system performs its function and the latter states that the system functions do not lead to an accident. A system may be reliable but unsafe. An example of such a system is an aircraft avionics system which continues to operate under adverse conditions such as component failure, yet directs a pilot to fly the aircraft on a collision course with another aircraft. The system itself may be reliable; its operation, however, leads to an accident. The system would be considered safe (in this case) if, on detecting the collision course, a new course was calculated to avoid the other aircraft. Similarly, a system may be safe but unreliable. For example, a railroad signalling system may be wholly unreliable but safe if it always fails in the most restrictive way; in other words, whenever it fails it shows "stop." In this case, the system is safe even though it is not reliable.

2.4 Software Need Not Be Perfect

A common theme running through the literature is that software need not be perfect to be safe. In order to make some sense of this view, we need to understand what is meant by perfection. Typically, we consider software to be perfect if it contains no errors, where an error is a variance between the operation of the software and the user's concept of how the software should operate. (We use the term "user" here to mean either the operator or designer or procurer of the software.) This notion of perfection considers all errors equal; thus, any error (from a spelling mistake in a message to the operator to a gross divergence between actual and intended function) means that the software is imperfect.

However, from a safety viewpoint, only errors that cause the system to participate in an accident are of importance. There may be gross functional divergence within some parts of the system, but if these are masked, or ignored by the safety components, the system could still be safe. As an example, consider a nuclear power plant using both control room software and protection software. The control room software could, potentially, contain many errors, but as long as the protection system operates, the plant will be safe. It may not be economical, it may never produce any power, but it will not be an agent in an accident. Even within a system such as the protection system, some bugs can be tolerated from the strictly safety viewpoint. For example, the protection system might always attempt to shutdown the reactor, regardless of the condition of the reactor. The system is not useful, it contains gross functional divergence, yet it is safe. This should be contrasted with a protection system that never attempts to shut down the reactor regardless of reactor condition. This system also contains gross functional divergence and is unsafe.

The view that software need not be perfect to ensure safety of the entire system means that developers and analysts of safe software can concentrate their most detailed scrutiny on the safety conditions and not on the operational requirements. Indeed, it is commonly assumed that other parts of the system are imperfect and may not behave as expected.

2.5 Safe Software Is Secure and Reliable

We have already discussed the differences between safety and reliability, but it should be clear that there are also distinct differences between safety and security. Safety does depend on security and reliability. Neumann discusses hierarchical system construction for reliability, safety, and security [169]. He also describes a hierarchy among these concepts. Essentially, security depends on reliability and safety depends on security (hence also reliability).

A secure system may need to be reliable for the following reason. If the system is unreliable, it is possible that a failure could occur such that the system's security is compromised. When determining whether a system is secure, the analyst makes assumptions about atomicity operations. If it is possible for the system to fail at any point, then the atomicity assumption may no longer hold and the security analysis of the system will be invalidated. Of course, it is possible for very carefully designed systems to be secure and unreliable, though the analysis for such systems will be harder than the analysis for reliable systems.

The safety critical components of a system need to be secure since it is important that the software and data cannot be altered by external agents (software or human). If the data or software can be altered, then the executing components will no longer match those that were analyzed and shown to be safe; thus, we can no longer rely on the safety critical components to perform their function. This may, in turn, compromise system safety.

It is obvious that, for some systems, safety depends on reliability. Such systems require the software to be operational to prevent mishaps: in other cases, it is possible to build systems where a failure of the software still leads to a safe system. In the case of non fail-safe software, if the safety system software is unreliable then it could fail to perform at any time, including the time when the software is needed to avoid a mishap.

2.6 Software Should Not Replace Hardware

One of the advantages of software is that it is flexible and relatively easy to modify. An economic advantage of software is that once it has been developed, the reproduction costs are very low. Hardware, on the other hand, may be quite expensive to reproduce and is, in terms of production costs, the most expensive part of a system. (For development costs, current wisdom indicates that the reverse is true, that the software development cost outweighs the hardware development cost.) Thus, from an economic viewpoint, there is considerable temptation to replace hardware components of a system with software analogs. However, there is a danger to this approach that leads to unsafe systems.

Hardware obeys certain physical laws that may make certain unsafe behaviors impossible. For example, if a switch requires two keys to be inserted before the switch can be operated, then both keys must be present before the switch can be operated. A software analog of this system could be created and indeed, with a relatively simple system, we may be able to convince ourselves of its correctness. However, as the software analogs become more complex, the likelihood of a possible failure increases and the software may fail permitting (in the case of our example) the software analog switch to be operated without either of the key-holders being present.

A concrete example of this behavior, taken from Leveson and Turner [141], is the Therac 25 radiation treatment machine. A predecessor to the Therac 25, the Therac 20, had a number of hardware interlocks to stop an undesirable behavior. Much of the software in the Therac 25 was similar to that of the Therac 20 and the software in both cases contained faults that could be triggered in certain circumstances. The Therac 25 did not have the hardware interlocks and where the Therac 20 occasionally blew fuses, the Therac 25 fatally irradiated a number of patients.

Furthermore, hardware fails in more predictable ways than software, and a failure may be foreseen by examining the hardware—a bar may bend or show cracks before it fails. These indicators of failure may occur long enough before the failure that the component may be replaced before a failure leading to a mishap occurs. Software, on the other hand, does not exhibit physical characteristics that may be observed in the same way as hardware, making the failures unexpected and immediate; thus, there may be no warning of the impending failure.

The concerns raised above are leading to the development of systems with both software and hardware safety components. Thus, the components responsible for accident avoidance are duplicated in both software and hardware, the hardware being used for gross control of the system and the software for finer control. An example, taken from a talk by Jim McWha of Boeing, is that of the Boeing 777. The design calls for a digital system to control the flight surfaces (wing flaps, rudder, etc.). However, there is a traditional, physical system in case of a software failure that will permit the pilot to operate a number (though not all) of the flight surfaces with the expectation that this diminished level of control will be sufficient to land the aircraft safely.

2.7 Development Software Is Also Safety Critical

Safety analysis of a system is performed on a number of artifacts created during the development of the system. Later stages in the development need not be analyzed under the following circumstances:

1. The analysis of the current stage of the development shows that a system performing according to the current description is safe.
2. There is certainty that any artifacts created in subsequent development stages precisely conform to the current description.

The earlier a system can be analyzed for safety with a guarantee that the second condition will be met, the more cost effective will be the overall development as less work will need to be redone if the current system description is shown to be unsafe. The disadvantage is, of course, that the earlier the analysis is performed, the greater the difficulty of achieving the second condition. Typically, the lowest level of software safety analysis performed will be at the level of the implementation language, whether it be in an assembly language or a high level language. In either case, the analyst is trusting that the assembler or the compiler will produce an executable image that, when executed on the appropriate target machine, has the same meaning as the language used by the analyst. Thus, the assembler or compiler may be considered to be safety critical. This is so because if the executing code does not conform to the analyzed system there is a possibility that the system will be unsafe.

Another part of the development environment that is critical is the production system. The analyst must ensure that the system description that has been shown to be safe is the exact same version as delivered to the system integrators. It is unsafe for an analyst to carefully analyze one version of the software if another version is delivered. Thus, certain parts of the development environment become critical. It is important that trusted development tools are used to develop the software for safety-critical systems.

3 Hazard Analysis Techniques

There are two aspects of the effort to performing a hazard check of a system; hazard identification and hazard analysis. Although these will be presented as separate topics, giving the impression that first the analyst performs all hazard identification and subsequently analyzes the system to determine whether or not the hazards can occur or lead to a mishap, the two activities may well be mixed. The general approach to hazard analysis is first to perform a preliminary hazard analysis to identify the possible hazards. Subsequently, subsystem and system hazard analyses are performed to determine contributors to the preliminary hazard analysis. These subsequent analyses may identify new hazards, missed in the preliminary hazard analysis, that must also be analyzed.

3.1 Hazard Identification

There does not appear to be any easy way to identify hazards within a given system. After a mishap has occurred, a thorough investigation should reveal the causes and lead the system engineers to a new understanding of the system hazards. However, for many systems, a mishap should not be allowed to occur since the mishap's consequences may be too serious in terms of loss of life or property.

The only acceptable approach for hazard identification is to attempt to develop a list of possible system hazards before the system is built.

There is no easy systematic way in which all of the hazards for a system can be identified, though it should be noted that recent work of Leveson and others [100] may prove to be an appropriate way of determining if all of the safety conditions for the particular system have been considered. The best qualified people to perform this task are experts in the domain in which the system is to be deployed. Petroski [182] argues that a thorough understanding of the history of failures in the given domain is a necessary prerequisite to the development of the preliminary hazard list. However, this understanding of the history is not sufficient. The experts need to understand the differences between the new system and previous systems so that they can understand the new failure modes introduced by the new system.

The resources required to obtain an exhaustive list of hazards may be too great for a project. Instead, the project management must use some approach to ensure that they have the greatest likelihood of listing the system hazards. The obvious approach is to use "brainstorming," where the experts list all of the possible hazards that they envision for the system. Project management also needs some guidelines to know when enough preliminary hazard analysis has been done. One such guideline might be when the time between finding new hazards becomes greater than some threshold value. While this is no guarantee that all the hazards have been identified, it may be an indication that preliminary hazard analysis is complete and that other hazards, if they exist, will have to be found during later phases of development. An alternative may be to use a consensus-building approach so that the experts agree that they have collected sufficient potential hazards for the preliminary hazard list. Approaches such as the Delphi Technique or Joint Application Design (JAD) may be employed.

3.1.1 The Delphi Technique

One of the older approaches to reaching group decisions is that of the Delphi Technique [60]. This method was created by the Rand corporation for the U.S. government and remained classified until the 1960s. The rationale for the development of the Delphi Technique was that there were many situations in which group consensus was required where the members of the group were separated geographically and it was not possible to get all members of the group together for a regular meeting. The method was originally designed for forecasting military developments, however, it may be used for any situation where group consensus is required and the group may not be brought together.

The basic approach is to send out a questionnaire to all members of the group that enables them to express their opinions on the topic of discussion. After the responses to the questionnaire have been received by the coordinator, the opinions are reproduced in such a way that the author's identify is obscured and the opinions are collated. The collated opinions are sent out to the experts who may agree or disagree in writing with the opinions and justify any outlying opinions. The expectation is that after a number of rounds of anonymous responses, the group will converge to produce some consensus decision. The group opinion is defined as the aggregate of individual opinions after the final round.

The key idea behind the Delphi Technique is that the opinions are presented anonymously and that the only interaction between the experts is through the questionnaires. The idea is that one particularly strong personality cannot sway the opinion of the entire group through force of will; rather, the group opinion is formed through force of reason. The Delphi Technique overcomes the issue of group consensus when the group is unable to attend a meeting where a method such as Joint Application Design might be employed. However, the nature of the Delphi Technique makes for slow communication and it may take several weeks to arrive at consensus. The use of electronic mail, a technology far newer than the Delphi Technique, may help overcome this problem.

3.1.2 Joint Application Design

Joint Application Design (JAD) was first introduced by IBM as a new approach to developing detailed system definition. Its purpose is to help a group reach decisions about a particular topic. Although the original purpose was to develop system designs, JAD may be used for any meeting where group consensus must be reached concerning a system to be deployed.

For JAD to be successful, the group must be made up of people with certain characteristics. Specifically, these people must be skilled and empowered to make decisions for the group they represent. Additionally, it is important for the right number of people to be involved in a

JAD session. Conventional wisdom suggests that between six and ten is optimum. If there are too few people then insufficient viewpoints may be raised and important views may therefore be lost. If there are too many people, some may not participate at all.

A JAD session is led by a facilitator who should have no vested interest in the detailed content of the design. The facilitator should be chosen for reasons of technical ability, skills in communication and diplomacy, and for the ability to maintain control over a group of people that may have conflicting views. It is recommended that a JAD session takes place in a neutral location so that no individual or group of people feels intimidated by the surroundings. A further advantage is that there should be fewer interruptions than if the meeting were held at the offices of one or more of the attendees.

JAD requires an executive sponsor, some individual or group of people who can ensure the cooperation of all persons involved in the system design and development.

It is important for the ideas presented by the group to be captured immediately and to develop a group memory. For JAD to operate optimally, ideas should become owned by the group rather than individuals, so it is recommended that any ideas be captured by the facilitator and displayed for all to see. This does have the disadvantage that the facilitator can become a bottleneck. There should be well-defined deliverables so that the facilitator can focus the meeting and ensure that the group makes progress.

3.1.3 Hazard and Operability Analysis

This form of analysis, also known as operating hazard analysis [145] or operating and support hazard analysis, applies at all stages of the development life cycle and is used to ensure a systematic evaluation of the functional aspects of the system.

There are two steps in the analysis. First, the designers identify their concepts of how the system should be operated. This includes an evaluation of operational sequences, including human and environmental factors. The purpose of this identification is to determine whether the operators, other people, or the environment itself, will be exposed to hazards if the system is used as it is intended. The second step is to determine when the identified conditions can become safety critical. In order for this second step to be performed, each operation is divided into a number of sequential steps, each of which is examined for the risk of a mishap. Obviously, the point in the sequence where an operation becomes safety critical varies from system to system, as it is dependent on the particular part of the operation, the operation itself, and the likelihood of a fault occurring in that step. The data generated from the analysis can be organized into tables indicating the sequence of operations, the hazards that might occur during those operations, and the possible measures that might be employed to prevent the mishap.

Hazard and operability analysis is an iterative process that should be started before any detailed design. It should be continually updated as system design progresses.

3.1.4 Summary

Both the Delphi Technique and JAD are approaches to obtaining group consensus on some topic. Although neither of these techniques was designed for determining the preliminary hazard list, it is clear that they can be used as a formal moans for capturing an initial list cf potential system hazards. Participants could be drawn from development and regulatory organizations the facilitator should be drawn from some neutral organization.

Hazard and Operability analysis provides a structured approach to the determination of hazards and may be used as the basis for the decision-making process.

3.2 Hazard Analysis

The purpose of hazard analysis is to examine the system and determine which components of the system may lead to a mishap. There are two basic strategies to such analysis that have been termed inductive and deductive [215]. Essentially, inductive techniques, such as event tree analysis and failure modes and effects analysis, consider a particular fault in some component of the system and then attempt to reason what the consequences of that fault will be. Deductive techniques, such as fault tree analysis, consider a system failure and then attempt to reason about the system or component states that contribute to the system failure. Thus, the inductive methods are applied to determine **what** system states are possible and the deductive methods are applied to determine **how** a given state can occur.

3.2.1 Fault Tree Analysis

Fault tree analysis is a deductive hazard analysis technique [215]. Fault tree analysis starts with a particular undesirable event and provides an approach for analyzing the causes of this event. It is important to choose this event carefully; if it is too general, the fault tree becomes large and unmanageable; if the event is too specific then the analysis may not provide a sufficiently broad view of the system. Because fault tree analysis can be an expensive and time-consuming process, the cost of employing the process should be measured against the cost associated with the undesirable event.

Once the undesirable event has been chosen, it is used as the top event of a fault tree diagram. The system is then analyzed to determine all the likely ways in which that undesired event could occur. The fault tree is a graphical representation of the various combinations of events that lead to the undesired event. The faults may be caused by component failures, human failures, or any other events that could lead to the undesired events (some random event in the environment may be a cause). It should be noted that a fault tree is not a model of the system or even a model of the ways in which the system could fail. Rather it is a depiction of the logical interrelationships of basic events that may lead to a particular undesired event.

The fault tree uses connectors known as *gates* which either allow or disallow a fault to flow up the tree. Two gates are used most often in fault tree analysis: the *and* and *or* gates. For example, if the *and* gate connector is used, then all of the events leading into the *and* gate must occur before the event leading out of the gate occurs.

The *and* gate (Figure 3-1) connects two or more events. An output fault occurs if all of the input faults occur.

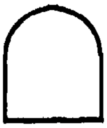


Figure 3-1: *And* Gate

Comparable to the *and* gate is the *or* gate (Figure 3-2) which connects two or more events into a tree. An output fault occurs from an *or* gate if any of the input faults occur.

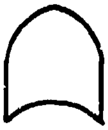


Figure 3-2: *Or* Gate

Other gates that may be used in fault tree analysis are *exclusive* or, *priority and*, and *inhibit* gates. These gates will not be used in this report and will not be explained further; a full description, however, may be found in the fault tree handbook [215].

Gates are used to connect events together to form fault trees. There are a number of types of events that commonly occur in fault trees.

The *basic* event (Figure 3-3) is a basic initiating fault and requires no further development.

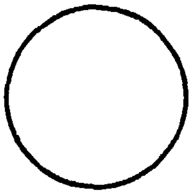


Figure 3-3: ***Basic*** **Event**

The *undeveloped* event symbol (Figure 3-4) is used to indicate an event that is not developed any further, either because there isn't sufficient information to construct the fault tree leading to the event, or because the probability of the occurrence of the event is considered to be insignificant.

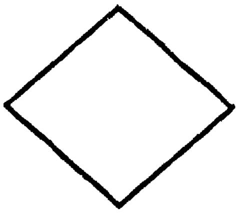


Figure 3-4: ***Undeveloped*** **Event**

The *intermediate* event symbol (Figure 3-5) is used to indicate a fault event that occurs whenever the gate leaoing to the event has an output fault. Intermediate events are used to describe an event which is the combination of a number of preceding basic or undeveloped events.

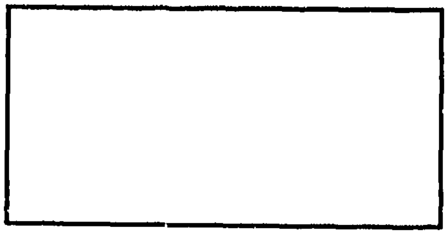


Figure 3-5: ***Intermediate*** **Event**

All of the events will generally contain text describing the particular fault that the event symbol represents.

The basic elements of a fault tree are gates and events. These may be tied together to form fault trees. As an example, consider the following simple fault tree in Figure 3-6. We wish to create a fault tree for the undesirable event of a car hitting a stationary object while driving on a straight road. As can be seen from the tree, the undesirable event is represented as an intermediate event at the top of the tree. We have chosen two possibilities, either of which could lead to the top event; these are that the driver doesn't see the object, or the car fails to brake. We could have added a third possibility, that the driver applied the brakes too late, however, we did not do so in this example. We considered possible causes for the driver failing to see the object. These might be that the object was on the road just around a corner, which has been represented as an undeveloped event, or that the driver is asleep at the wheel, a basic event of the system. We chose to represent the possibility that the object was around a corner as an undeveloped event since this is unlikely given that the road is a long straight road (from the problem definition); however, there is a possibility that the object is on the road at the very start of that road and that the driver must first negotiate a corner before getting onto the road. There might be many other possibilities why the driver doesn't see the object; these include fog, the driver being distracted, the driver being temporarily blinded, the car travelling at night without lights, etc. When we considered reasons why the car failed to brake, we listed brake failure or ineffective brakes as possibilities. Brake failure was represented as an undeveloped event, not because it is an insignificant event, but because we have insufficient information as to why brakes fail—domain expertise is required to further elaborate this event. We developed the ineffective event into two events, both of which must occur for the brakes to be ineffective: the car must be travelling too fast and the brakes must be weak.

As can be seen, the development of a fault tree is a consideration of the possible events that may lead to a particular undesirable event. Domain expertise is necessary when developing fault trees since this provides the knowledge of how similar systems have failed in the past. Knowledge of the system under analysis is necessary since the particular system may have introduced additional failure modes or overcome failures in previous systems.

Fault tree analysis was initially introduced as a means of examining failures in hardware systems. Leveson and Harvey extended the principle of fault tree analysis to software systems [118]. Fault trees may be built for a given system based on the source code for that system. Essentially, the starting place for the analysis is the point in the code that performs the potentially undesirable outputs. The code is then analyzed in a backwards manner by deducing how the program could have got to that point with the set of values producing the undesirable output. For each control construct of the programming language used, it is possible to create a fault tree template that may be used as necessary within a fault tree. The use of templates simplifies the question of "How can the program reach this point" and reduces the possibility of error in the analysis.

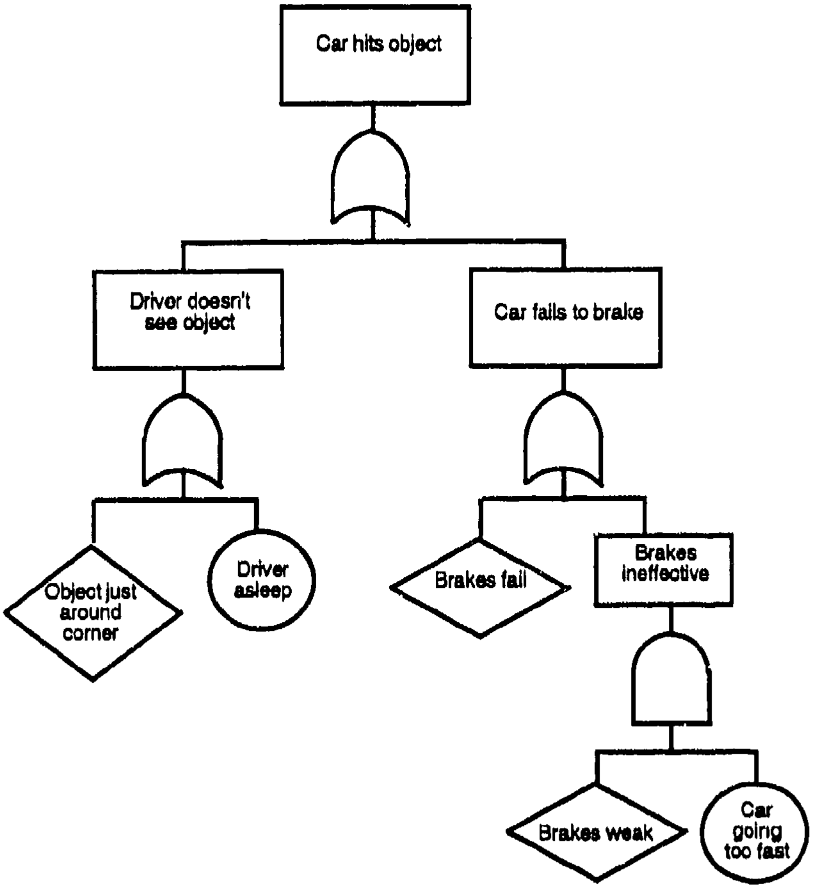


Figure 3-6: Example Fault Tree for a Car Crash

Fault tree analysis need not be applied solely to programming language representations of the system. Any formally defined language used to represent the system may be analyzed using fault trees and templates may be created for notations used at different stages of the system development life cycle. Later in this report, we discuss the application of software fault tree analysis to system specification.

3.2.2 Event Tree Analysis

Event tree analysis is an inductive technique using essentially the same representations as fault tree analysis. Event trees may even use the same symbols as fault trees. The difference lies in the analysis employed rather than the representation of the trees.

The purpose of event tree analysis is to consider an initiating event in the system and consider all the consequences of the occurrence of that event, particularly those that lead to a mishap. This is contrasted with fault tree analysis which, as has been described, examines a system to discover how an undesirable event could occur and eventually leads back to some combination of initiating events necessary to cause the failure of the system. Thus, event tree analysis begins by analyzing effects while fault tree analysis begins by analyzing potential causes.

The approach taken is to consider an initiating event and its possible consequences, then for each of these consequential events in turn, the potential consequences are considered, thus drawing the tree. It may be that additional events are necessary for an intermediate event to occur and these may also be represented in the tree.

The initiating events for event tree analysis may be both desirable and undesirable since it is possible for a desirable event to lead to an undesirable outcome. This means that the choice of initiating events is the range of events that may occur in the system. This may lead to difficulty in deciding which events should be analyzed and which should not in an environment where only limited resources are available for safety analysis.

Event tree analysis is forward looking and considers potential future problems while fault tree analysis is backward looking and considers knowledge of past problems.

Event tree analysis is not as widely used as fault tree analysis. This may be in large part due to the difficulty of considering all of the possible consequences of an event or even the difficulty of choosing the initiating event to analyze. One reason for this is that trees may become large and unmanageable rapidly without discovering a possible mishap. Much analysis time may be wasted by considering an event tree from a given event, such as the failure of a sensor, when that event may never lead to a mishap. This may be contrasted with fault tree analysis which is directed toward the goal of a specific failure.

In systems where there is little or no domain expertise available (that is, wholly new systems), event tree analysis may play a valuable role since the consequences of individual component failures may be analyzed to determine if a mishap might occur, and what that mishap might be. In systems with past history, fault tree analysis would appear to be a better analysis technique.

3.2.3 Failure Modes and Effects Analysis

Failure Modes and Effects Analysis (FMEA) [53] is another inductive technique and attempts to anticipate potential failures so that the source of those failures can be eliminated. FMEA consists of constructing a table based on the components of the system and the possible failure modes of each component. FMEA is not an additional technique that engineers have to learn, but rather a disciplined way of describing certain features (the failure modes) of the components and the effects these features have on the entire system.

The approach used is to create a table with the following columns: component, failure mode, effect of failure, cause of failure, occurrence, severity, probability of detection, risk priority number, and corrective action. Table 3-1 is an example FMEA for part of an engine mounting system. We have considered only the single tie bracket component, and only a few of the possible ways in which the bracket may fail.

Table 3-1: Example Failure Modes and Effects Analysis Table

Component	Failure Mode	Effect of Failure	Cause of Failure	Occurrence	Severity	Probability of Detection	Risk Priority Number	Corrective Action
Tie Bar Bracket	Bracket fractures	Stabilizing function of tie bar removed. All engine motion transferred to mountings	Inadequate specification of hole to edge distance	1	7	10	70	Test suitability of specification
	Bracket corrodes	As above	Inadequate specification for preparation of bracket	1	5	10	50	Test suitability of specification
	Fixing bolts loosen	As above	Bolt torque inadequately specified	5	5	8	200	Test for loosening
			Bolt material or thread type inadequate	1	5	10	50	Test suitability of specification

For each component, a list of the possible failure modes is created. These failure modes are used to populate the second column of the table. The effects of each failure are considered and entered into the third column. Although the existing literature does not indicate that it should be done, it would seem that use of event tree analysis may help in determining the possible effects of the component failure. The potential causes of the failure mode are listed in the fourth column of the table and similarly, though not mentioned in the literature, it would seem that fault tree analysis might be the appropriate technique for determining causes of the component failure.

The engineer is then required to enter a value indicating the frequency of occurrence of the particular cause of the failure mode. For existing hardware components, statistical data may exist to accurately predict failure. However, in most cases, particularly for software, the engineer will have to use knowledge and experience to make a best estimate of the value. The values for the occurrence field should lie between 1 and 10, with 1 being used to indicate very low probability of occurrence and 10 a near certainty.

Based on the determination of the possible effects of the failure mode, the engineer must estimate a value that indicates the severity of the failure. Note that this is independent of the probability of occurrence of the failure, but is simply used as an indicator of how serious the failure would be. Again, a value between 1 and 10 is used, with 1 being used to indicate a minor annoyance and 10 a very serious consequence.

The next field is the detection of failure field. Here the engineer must estimate the chance of the failure being detected before the product is shipped to the customer. It may well be that for software systems, this field will be estimated based on the quality of the testing process and the complexity of the component. Again a score between 1 and 10 is assigned, with 1 indicating a near certainty that the fault will be detected before the product is shipped and 10 being a near impossibility of detection prior to shipping.

The risk priority number is simply the product of the occurrence, severity, and failure detection fields and provides the developers with a notion of the relative priority of the particular failure. The higher the number in this field, the more serious the failure—leading to indications of where more effort should be spent in the development process.

The final field of the FMEA table is a description of potential corrective action that can be taken. It is unclear whether this field has any meaning in software systems and further investigation should take place to determine if any meaningful information can be provided by the safety engineer. It may be that for software components, corrective action will be the employment of techniques such as formal methods for fault reduction or fault tolerance techniques for fault detection and masking.

A closely related approach is the use of Failure Mode, Effects and Criticality Analysis which performs the same steps as FMEA, but then adds a criticality analysis to rank the results. The FMEA described does provide a way of ranking results, however the FMECA provides a more formal process for performing the criticality analysis.

3.3 Summary

The process of performing a safety analysis of a system is time consuming and employs many techniques all of which require considerable domain expertise. It is clear that for the safest possible systems, the best available staff should be used for the safety analysis.

There would appear to be two approaches that can be taken:

1. Create a list of all hazards and for those with a sufficiently high risk perform fault tree analysis indicating which components are safety critical. Then for those components, continue to apply hazard analysis techniques at each stage of development.
2. Perform an FMEA for all components of the system, potentially using fault tree and event tree analysis to determine causes and effects of a component failure respectively. Employ the best development techniques (usually more expensive) on those components with an unacceptably high criticality factor.

4 Development Techniques for Safety-Critical Software

We have discussed techniques that help the system developers determine the safety critical components of the system. It is important that the development process be one that does not introduce new failures that may lead to a system mishap. This chapter outlines the techniques most commonly discussed in the literature that avoid the introduction of errors into the system during the development process. Note, though, that at each level of representation it is possible to employ hazard analysis techniques such as fault tree analysis. Such an analysis is not excluded by the techniques described in this chapter, nor does it replace the need for the techniques described. The development process is complemented by the analysis of each development and for the safest systems, both approaches should be used.

There is a widely held belief that formal methods should form part of the development process for safety-critical systems. Experiments have been reported in a number of conferences (such as Daniels [47]) which support this belief. Thus, each section that follows will contain some discussion concerning the relevance of formal methods to the particular development topic.

It should be noted, though, that researchers and developers do not see formal methods as the only technique that should be employed; rather, formal methods should be used in conjunction with existing approaches to system development. It is the combination of techniques that will lead to safer systems.

4.1 Requirements

The requirements for a system are generally presented in terms of natural language description of the function of the system, desired performance characteristics, predetermined design decisions such as the use of particular hardware or software packages, and many other non-functional characteristics such as the maintainability of the system.

Because the natural language representation is often ambiguous and incomplete, only limited analyses may be performed. Typically, the first step is to represent the requirements in a notation that is not ambiguous and may be analyzed. This is the process of specification; the result is a specification of the behavior of the system. Given that the specification is in a notation that may be unfamiliar to the system procurers, the specification should be validated with the procurers to ensure that the specification is a true representation of the intended system.

4.1.1 Specification and Analysis

There are many notations that may be used to specify systems, and these notations have varying levels of formality. Standards such as the U.K. Ministry of Defence Standard MOD 00-55 (see Section 5.1.1) require that the specification be written in a formal notation. Other standards offer other alternatives. We will concentrate on formal notations since these offer the greatest opportunity for analysis.

There is no simple way to formalize the requirements to produce the specification. The specifiers must read the requirements and translate them into statements written in the formal notation. It is important for the specifiers to be expert in the domain in which the system is to be deployed. The specification may be analyzed for inconsistencies and, to a limited extent, incompleteness (such an analysis depends on the expertise of the analysts rather than the specification technique used).

There are a number of different types of formal specification technique that may be used. Some notations are better suited to the specification of concurrency while others are better suited to sequential systems. The choice of notation should depend on the system being specified and the expected expertise of the developers who will have to read and understand the specification. Leveson and others have described the use of Petri-nets [119] and also the use of state machines [98] as appropriate specification notations that may be used to model the system and analyze the system for unsafe behaviors. MOD 00-55 lists eight notations that may be used for specification.

Melhart describes the use of Statecharts for the specification of the system properties and then indicates how a fault tree analysis may be performed on the Statechart representations of the system [159].

What is common to each of the approaches is that the requirements are formalized using a mathematically-defined notation. Subsequently the formal representation of the requirements is analyzed and undesirable properties are removed.

4.1.2 Validation

One of the hard issues to handle in the formalization of the requirements is ensuring that the specification describes the desired system. It is certain that the specification describes a model of some system, but the question remains as to whether the model accurately represents the desired system. There is no way to formally prove that the specification is a model of the desired system. (It should be noted that this is true whether or not the specification is written in a formal notation.)

The specification needs to be validated with the people who developed the requirements. The validation process may take a number of forms. Some specifications are executable and the system specification may be used to validate behavior. This is done by running the specification on appropriate sets of inputs and determining if the behavior of the specification is consistent with the desired behavior for the system. Some specifications are not executable. However, for such specifications, it is possible to use the specification as a model to predict how the system will behave given that the initial state and inputs to the system are known. This can be checked against the notion of the correct state of the system for validity. Usually, the prediction is done by a mathematical proof and by creating a formal description of the resulting state. This is best validated by interpreting the mathematics back into a natural language description of the state.

The specification may be compared with a mathematical model of any existing standards that apply to the domain in which the system is to be deployed and it may be possible to demonstrate that a system conforming to the model described by the specification meets (or fails to meet) the definitions of safety described in any appropriate standards.

4.2 Design

Assuming that a formal specification has been created and validated with respect to system safety, design of the system must be performed with two considerations in mind. The first consideration should be that the design does not create new system hazards by adding unintended function to the system or eliminating function described in the specification; the second, that the representation of the system comes closer to an implementation, filling in detail as necessary. This latter consideration is the usual design consideration and we will not discuss it further in this document. Note also that, as for specifications, safety analysis may be performed during the design process [38].

Looking at the correctness consideration for the design process, it is clear that the specification must be written formally. Otherwise, the design, even if presented in a formal notation, cannot be checked against the specification. (This is the same problem as checking a formal specification against informally represented requirements.)

The two approaches to design are:

1. To perform the design using some appropriate design principles and notation and then represent the result of the design process in a formal notation and prove that the design satisfies the specification.
2. To successively transform the specification using design principles as a guide to the selection of the transformations and demonstrate the correctness of each transformation.

The advantages to the first approach are that the more familiar design notations may be employed. However, there are a number of disadvantages.

1. The transformation from the design notation into the formal notation may lead to errors that will only be detected when the formal representation of the design is compared to the specification. A corollary is that all of the design work will have been performed before a formal check that the design satisfies the specification can be performed. If an error is introduced in the design process much work may have to be redone.
2. The proof that the design satisfies the specification will be hard and complicated since the two representations of the system will be dissimilar.
3. If corrections or enhancements are to be made, it may be less clear how to correct all of the design documentation, since it may not be possible to infer from which parts of the design particular pieces of the implementation have been derived. The use of different notations complicates tracking of specification through design.

The approach of successive transformation overcomes the disadvantages outlined above. However, it also has disadvantages.

1. This is still an immature technology and although there have been a number of publications on the topic [165], it is, at the time this report is being written, unproven in large scale applications. We can assume that, as time passes and the transformational approach is employed more frequently, this disadvantage will evaporate.
2. There will be many more representations of the system, each of which will be similar to the preceding and successive representations. This places a much greater burden on the development environment, particularly with respect to tracking the connections between pieces. Some experiments have indicated that a hyper-text-based environment may overcome this problem.

4.3 Implementation

It may be argued that the most important concern of implementation is conformance between the executing code and the design, that is, that the executing code has the same semantics as the lowest level of design. There are a number of aspects of implementation that affect conformance: the development tools used, formal verification of implementation, and runtime checking.

4.3.1 Development Tools

We have already touched on the idea that the development environment is, to some extent, safety critical. It is important that the software that has been checked and found to be safe is the software that is built and delivered. One aspect of the development environment of particular importance is the version management system. Typically, many analyses are performed at the implementation level on the code that is compiled into the system. Analyses such as code reviews and software fault tree analysis make the assumption that the code that is used to build the software is the code that has been analyzed. There are two consequences of this assumption.

1. That a version management system exists and is used so that the system integrators may have absolute confidence that the code that was analyzed is the code used to perform the system build. A similar potentially hazardous situation can occur if the code is built and tested and then small changes are made and the resulting system is not put through the same testing process. Examples of such failures occurred in 1991 when both AT&T and a number of the Bell telephone exchanges failed and phone connections could not be made. In both cases, seemingly minor changes were introduced into the system after the testing process had been completed that introduced wholly new failure modes for the systems.
2. That the compiler, assembler and linker produce an executable image that has the same meaning as assumed at the level of the code. The implication here is that the development tools should be formally verified and that the code should be run on verified hardware. This is the idea behind the Computational Logic Inc. stack and the ProCoS project.

4.3.2 Formal Verification

One very strong approach to demonstrating conformance between the implementation and the design is to formally verify that the implementation has exactly the same meaning as described in the design.

Formal verification requires that the design be expressed in a formal notation (that is, a notation with mathematically defined semantics) and that the semantics of the implementation language are also formally defined. Note that only the semantics of the language constructs that are used in the implementation need to be formally defined. Thus, it is possible to use a subset of a language for which full formal semantics do not exist as long as the semantics of the constructs in that subset are defined.

The process of formal verification, then, is one of proving a correspondence between the statements of the program and the statements of the design. In many cases, the design is expressed as a number of program pieces (module, procedure, function, sequence of statements) with conditions on the piece describing the input and output states for that piece. These descriptions are generally stated in terms of a predicate describing all of the input states for which the program piece is expected to operate and a predicate on the output states showing the relationship between the input states and the output states. Then, it is possible to examine the code and construct a proof that the statements in the appropriate program piece do implement the relationship between the input and output states for all the inputs described by the input predicate.

It should be stated that the proof of correctness is not a trivial task and that mistakes may be made. Thus, the use of tools to assist in the proof is an important part of the process. Again, this approach depends on the compilation tools as it makes the assumption that the executable image has exactly the same meaning as the meaning implied by the statements of the program fragments using the formal semantics of those statements.

4.3.3 Runtime Checking

Another approach to ensuring conformance between the implementation and the design is to check the operation of the system dynamically. There are two classes of approach that may be used to check the runtime behavior of the system: the development of self-checking code or the development of an independent monitor. In both cases, the checking part of the system will, if the system deviates from the expected behavior, take steps to avoid a hazard.

There are two approaches that might be used to create self-checking code: have the developers insert additional checks into the code or have the compiler insert the checks.

1. It is certainly possible for the developers to insert additional checks into the code, which essentially monitor the state of the system, and if an erroneous state is detected to take actions to correct the state. The effectiveness of this approach, however, is questionable: as an experiment by Leveson and others [135] indicates that in many cases, the self-checking code failed to detect known errors.

2. The compiler may be used to insert checks into the code based on the design specification. An example of this approach is the ANNA language (Annotated Ada) [147]. Assuming that the design is represented in ANNA and the code is in Ada, then the compiler will insert runtime checks into the executable code based on the ANNA design specification. These checks will raise an exception (which would need to be trapped and appropriate hazard-avoiding action taken) when the state of the system is different from the ANNA description. The advantage of this approach is that the checks are inserted automatically at every point in the code where there is an applicable ANNA description (for example, procedure entry and exit or even at the level of a variable changing value). The disadvantage of this approach is that it may add considerable processing time to the execution of the system.

The other approach described in the literature is that of a monitor that acts independently of the software and checks the outputs from the software. An example of the monitor approach is described by Leveson in papers on Murphy [124]. If those outputs are at variance with the monitor, then the monitor may either substitute its own outputs or invoke some other piece of software that will return the system to a safe state. The monitor approach may be used to examine just the outputs of the software or may be extended to the state of the entire system.

The operation of the monitor is very important in this type of approach and the developers must be able to guarantee that the monitor will always act correctly. That is, the computations within the monitor must be correct with respect to the expectations of the monitor function and also that any hazard-avoiding action the monitor takes must operate correctly. If the monitor or the avoidance actions fail to function correctly, the system may still be hazardous due to failures in the software, indeed, the system may be more hazardous since the monitor could take over control when the monitor is in an erroneous state and lead to the occurrence of a mishap.

5 Standards

This chapter outlines the standards that pertain to the development of software for safety-critical systems. The chapter concludes with a section describing the possible negative effects that standards have on the development community.

5.1 MOD 00-55 & MOD 00-56

These two standards were produced by the U.K. Ministry of Defence (MOD) in 1991, though draft versions were available earlier. These standards are labelled as interim defense standards. This means that they are not yet in full force. The standards may evolve further before they are fully enforced. Essentially, they may be treated as a statement of intent, that is, that the MOD expects that at some time over the next five years, the standards as written, or a variation on these standards, will be enforced.

Although each of these documents is a standard in its own right, they are generally considered as a cooperating pair of standards. MOD 00-55 concerns the procurement of safety-critical software and MOD 00-56 concerns the hazard analysis and safety classification of computer hardware and software.

5.1.1 MOD 00-55

This standard [163] covers the procurement of safety-critical software for Ministry of Defence (UK) systems. Software is determined to be safety critical using safety integrity requirements that are determined according to MOD 00-56. The standard is in two parts, requirements and guidelines.

The standard describes a software development process where verification and validation are integral parts. Formal methods, dynamic testing and static path analysis are techniques required by the standard to achieve high levels of safety integrity. It is stressed that the standard only applies to safety-critical software. The requirements of the standard have three major sections: general, safety management, and software engineering practices requirements.

First, the general section introduces the standard, and the scope of the standard. It should be noted that the standard disclaims responsibility for liability even if the standard is followed completely. The general section also introduces definitions used throughout the document.

The second section details safety management. The standard requires various management practices to be employed in the development of safety-critical systems. One example is that a specific individual is required to be responsible for all safety issues throughout the life of the system. Another example is a requirement to identify safety critical components as soon as possible and, at each stage of design, a hazard analysis and safety risk assessment must be carried out (in accordance with MOD 00-56) to identify all potential failures that may cause new hazards. Associated with this requirement is a further requirement to establish the correct safety integrity level for all system components.

The standard requires that a risk analysis be performed that demonstrates that the techniques and tools described in the safety plan are appropriate to the type of system being developed and that development can be undertaken with acceptable risk to the project success. The risk analysis must be performed early in the project life cycle and be re-performed if any of the assumptions on which it is based change.

The standard requires that the verification and validation (V&V) team be independent of the design team. The V&V team must prepare a verification and validation plan to verify the safety-critical software by using dynamic testing and by checking the correctness of formal arguments presented by the design team.

There is a requirement for an independent safety auditor to be appointed. The position of the independent safety auditor is created under a separate contract and, if at all possible, the same individual is expected to act as independent safety auditor throughout the system development. The independent safety auditor must be commercially and managerially separate from the design team and will produce an audit plan at the start of the project that will be updated at the start of each subsequent project phase. The independent safety auditor oversees all work that influences safety integrity and periodically audits the project to ensure conformance to the safety plan.

A safety plan is to be developed in the earliest phases of the project, no later than the project definition phase, and is to be updated at each subsequent phase of the project. The safety plan:

- Shows the detailed safety planning and control measures that will be used.
- Contains descriptions of the management and technical procedures used for the development of the safety-critical software.
- Describes the resources and organizations required by standard.
- Identifies the key staff by name.

The standard requires that a safety records log be maintained which contains evidence that the required safety integrity level is achieved. The safety records log includes the results of hazard analyses, modelling reports, and the results of checking the formal arguments.

The standard describes requirements on documentation, deliverable items, configuration management: the requirements on certification and acceptance into service; and requires that the design team submit a safety-critical software certificate, signed by the design team and counter-signed by the independent safety auditor providing a clear, unambiguous and binding statement that the software is fit for use and conforms to the requirements of the standard.

The third section of the requirements describes the software engineering practices that are to be used by the developers of the safety-critical software. This section discusses issues relating to specification, design, coding and reuse.

The standard states that the first step in the design of safety-critical software is the production of a software specification from the software requirements specification. The software specification must include a specification using the notation of an approved formal method and an English commentary (including any appropriate engineering notations) on the specification. The design team is required to check the formal specification for syntactic and type errors using an approved tool. The team is also required to construct proofs showing that the specification is internally consistent. Further, the design team is required to validate the software specification by means of animating the formal specification. The animation is performed by the construction of various formal arguments and by the execution of an executable prototype derived from the software specification.

The standard requires that each step of the design be described using a design description. The design description comprises a formal description of the design using an approved formal method and an English commentary on the design. The standard requires that safety-critical software is designed so that it is easy to justify that the design meets the specification. This may mean using short, uncomplicated software and may inhibit the use of concurrency, interrupts, floating point, recursion, or a number of other aspects of programming. The design team is required to construct formal arguments demonstrating that the formal design satisfies the formal specification.

The standard requires that coding standards that lead to clear, analyzable code be used. The code must be analyzed using formal arguments and static path analysis. The implementation language used must have various characteristics including block structure, strong typing, a formally defined syntax and a well understood semantics. The design team must use a static path analysis tool to check control flow (including redundant code), data use and information flow. The team must also create formal arguments that prove that the code satisfies the formal design.

Much of the development requires the use of formal arguments. These may either be formal proofs or rigorous arguments, the former being a complete mathematical proof and the latter being the outline of a proof. (It is expected that formal proofs will be required most often but there are cases where a rigorous argument would be sufficient.)

There are requirements for the performance of dynamic testing and for reusing existing software. The latter requirements state that there must be agreement from the safety assurance authority, the program manager, and the independent safety auditor before the software may be reused. Further, if necessary, the design team may have to formally specify and verify the software being reused.

The standard requires that all tools used in the development of safety-critical software have a sufficient safety integrity level to ensure that they do not jeopardize the safety integrity of the safety-critical software. The development tools are assigned their integrity levels according to MOD 00-56.

The second part of MOD 00-55 is guidance on the requirements stated in the first part. It elaborates on the requirements to make the achievement and assessment of conformance to the requirements easier. Further, it provides technical background to the requirements.

The system should be designed so that the safety-critical software is isolated as fast as possible. This isolation minimizes the amount of software that has to be developed to the high levels of safety integrity. This is particularly important for software that does not implement a safety function, but whose failure, if it is tightly coupled with safety-critical software, may cause the safety-critical software to fail.

The standard suggests that the independent safety auditor should be a chartered engineer and have a minimum of five years' experience with safety-critical software and its implementation. The auditor should also be experienced in the methods that the design team proposes to use.

The standard offers guidance on the safety reviews which includes formal reviews such as Fagan inspections and various checks on the English commentaries, including spelling checks and checks for unexpected words in the document (i.e., words that pass the spelling check but are erroneous).

Criteria for the selection of the formal methods include: that the notation should be formally defined, that the method should provide guidance on strategies for building a verifiable design, that there are case studies in the literature demonstrating successful industrial use, and that courses, textbooks, and tools should be available.

Guidance is also offered on the use of the formal method. This includes guidance on checks to be performed on the formal specification as well as the generation and discharge of proof obligations through formal reasoning.

As stated in the discussion of the first part, the standard recommends that various programming techniques be avoided. The second part offers reasons why the particular programming techniques should be avoided. The general reason is that use of certain programming techniques makes some formal arguments harder to prove.

Guidance is offered on the type of programming language to be used. It is strongly suggested that a conventional procedural language be used rather than assembly language or other unconventional languages. There is guidance on how to perform static path analysis and how to review the results of that analysis.

Tools are discussed, as well as a scheme for tool integrity level. This is a four-level scheme, comparable to that of the safety integrity level scheme. Tools are also classed as transformational (such as a compiler), V&V (such as static path analyzers), clerical (such as editors), or infrastructure (such as the operating system or configuration management).

5.1.2 MOD 00-56

The purpose of this standard is to identify, evaluate and record the hazards of a system to determine the maximum tolerable risk from it, and to facilitate the achievement of a risk that is as low as reasonably practicable and below the maximum tolerable level [164]. This activity will determine the safety criteria and a reasonable and acceptable balance between the reduction of risk to safety and the cost of that risk reduction.

The standard uses a hazard log, which is described as the principle means for establishing progress on the resolution of intolerable risks. Using the hazard log, the standard helps the contractors determine whether the system can cause accidents, and, if the system is hazardous, if the risk from the system is tolerable. The hazard log also provides a mechanism to identify and track critical components. The hazard log is initiated during the initiation phase of the life-cycle and is updated both as hazards are discovered or as previously identified hazards have been eliminated or their associated risk reduced to a tolerable level. The hazard log is reviewed on a regular basis through the project life cycle.

The standard uses the notion of four classes of risk. These are determined by classifying each hazard in one of the four categories of accident severity (catastrophic, critical, marginal, and negligible), assigning one of six probability levels to the hazard and using a table to determine the risk class (intolerable, undesirable, tolerable—if the project safety review committee agrees, and tolerable) using normal project reviews.

The standard requires that for every identified hazard, a preliminary hazard analysis, system change hazard analysis, and a safety review must be performed. For hazards that are intolerable or undesirable an independent safety audit is required.

There are five approaches in decreasing order of preference to be used to reduce the risk associated with a hazard: re-specification, redesign, incorporation of safety features, incorporation of warning devices, and operating and training procedures.

The standard requires that the hazard classification depends not only on the system being developed, but also on the systems with which it will interact and the environment in which it will operate. If the system is used in a new environment or will interact with different systems, the analyses must be re-performed. The standard recognizes that many parts of the original analyses may still be useful in the new environment, however, they cannot be used as a substitute for analysis in the new environment.

The standard describes various forms of analysis to be performed. Preliminary hazard identification which may be initiated using data from similar systems and enhanced using a hazard and operability analysis or an alternative technique that satisfies certain criteria. Preliminary hazard analysis sets the boundaries for the system; it identifies the system hazards based on the preliminary hazard identification and the requirements. The results are recorded in the hazard log. Potential accidents are identified and categorized with a risk class. The system hazard analysis includes a functional analysis, which determines the hazards due to correct or incorrect function of the system, a zonal analysis, which considers the consequences of failures in adjacent systems; and component failure analysis, which requires a failure modes and effects analysis of the components. The contractors must also perform a system risk analysis which associates the identified hazards and possible accident sequences with their risk classes. The system risk analysis also requires that a failure probability analysis be carried out using fault tree analysis.

For all changes, the contractor is required to carry out a system change hazard analysis to determine that new hazards are not being introduced by the change or that existing hazards are not increased in risk.

Any properties of the system intended to remove the intolerable hazards and reduce other risks are known as safety features and must be documented in the specification and design documents.

The standard uses the concept of safety integrity level to classify functions according to how much effect the function has on system hazards. Tables are presented that help the contractors determine the safety integrity level for functions. The concept is also used to discuss high-level functions and the requirements on implementation of a high-level function with a particular safety integrity level from low-level functions with differing safety integrity levels.

The standard also makes requirements on the management of the program. There is a requirement that the program has a project safety engineer with sufficient seniority and authority to represent the development organization. The project safety engineer is responsible for all safety matters including signing statements of risk and component criticality. In addition, for any program with either catastrophic or critical hazards, an independent safety auditor must be appointed. The independent safety auditor must have full access to the results of hazard analyses and safety risk assessments. The auditor must be independent of the development organization and should not be changed during the development of the system without good reason.

The standard requires that all objects produced by hazard analysis or safety risk assessment, including relevant data, be held under configuration control to satisfy certain standards.

5.1.3 Summary

These two standards offer a very strong statement on the development of safety-critical software. Essentially, they state that until demonstrated otherwise, all software is assumed to be safety critical. Once software has been analyzed, it may be assigned a safety integrity level which determines the type of effort required for the development of that software. Software with a high safety integrity level must be specified, designed, and implemented using appropriate formal methods with formal proofs of correctness being presented between all levels of the software.

5.2 MIL-STD-882B

The MIL-STD-882B standard [54], developed by the US Department of Defense in 1977 and updated in 1984, requires that contractors establish and maintain a formal system safety program that ensures that: safety consistent with the mission is designed into the system; hazards associated with the system are identified, evaluated and eliminated or the associated risk is reduced to an acceptable level; uses historical data concerning failures; and records significant safety data for use in other systems.

Design requirements are described which state that the first aim of the contractors is to eliminate hazards or at least to reduce the risk associated with the hazard through the design process. The designers are required to isolate hazardous substances, components, or operations from the rest of the system. The designers are also required to design software-controlled or software-monitored functions to minimize the initiation of hazardous events or mishaps.

A hierarchy of safety procedures is described. In decreasing order of importance, these procedures are: designed for minimum risk, incorporate safety devices, provide warning devices, and develop procedures or training. Hazards are also categorized into four levels: Catastrophic, critical, marginal, and negligible. The standard states that for hazards that are either catastrophic or critical, unless a waiver is granted, a safer design or safety devices must be used to reduce the risk.

The standard recognizes that at the start of development quantitative values for the probability of an event occurring will not be available. Instead, qualitative values may be used, though the standard requires that a rationale for the choice of the probability level must be given. The qualitative probability levels are divided into five categories: frequent, probable, occasional, remote and improbable. The standard then describes three task areas concerned with program management and control, design and evaluation, and software hazard analysis.

The first task area, program management and control, includes a number of tasks. These tasks require the contractor to create a system safety program plan. This plan describes the tasks and activities of system safety engineering and system safety management. It describes the tasks of the system safety organization and authority it holds. The task area outlines the task that verifies that the safety organization has done its job correctly. Other tasks in the management area require that the contractor participate in any system safety groups that are formed. The task area also requires that the contractor maintain a hazard log that documents all hazards from the time the hazard is identified to the time when the hazard is eliminated or the associated risk is reduced sufficiently. Finally, the standard also requires that the contractor report the status of hazards at periodic intervals, and defines levels of qualifications that key safety engineers should hold.

The second task area describes tasks associated with performing various analyses for safety. The tasks required are those involved with the preparation of a preliminary hazard list by performing a preliminary hazard analysis which should be started either during concept exploration or as early as possible during development. The contractors are required to perform a subsystem hazard analysis which should include issues of safety relating to the possible modes of failure including those caused by reasonable human error or single-point equipment failure. The contractors are also required to perform system hazard analysis, operation and support hazard analysis, safety assessments and software hazard analysis. This latter analysis, though required in the design task area, is the focus of the third task area.

The third task area concentrates on software hazard analysis. The standard states that it is to be used in conjunction with other standards such as DOD-STD-2167 and MIL-STD-483A. The task area requires that the software be analyzed at the following levels: software requirements, top-level design, detailed design, and code level. Each of these analyses should check for input/output timing, multiple event, out of sequence event, wrong event, failure of event, inappropriate magnitude, adverse environment, deadlock, or hardware failure sensitivities. Further, at the code level, the contractor is required to analyze the software for implementation of the safety criteria, and for combinations of hardware or software or transient errors that could cause hazardous operation. The contractor must also perform flow analysis of the code, ensure that there is proper error default handling, and that there are fail-safe or fail-soft modes. The contractor is also required to perform software safety testing, a user interface analysis and, for every change to be made, a software change hazard analysis to determine the impact on safety of a proposed change.

In summary, this standard describes many tasks that contractors must perform. It should be noted that the standard does not describe specific techniques that must be used, rather it outlines the tasks and allows the program manager to choose techniques or, if the contractor chooses techniques, requires that the program manager approve of the techniques. The standard takes a systems view of safety and discusses the particular approaches to be taken if there is a significant software portion to the system.

In the rationale, the standard lists various techniques for ensuring safety. These include software fault tree analysis, software sneak analysis, code walk-throughs and Petri net analysis. The rationale recognizes that these techniques have different strengths and weaknesses and states that a thorough software hazard analysis will require the application of more than one of the techniques.

5.3 DO-178A & MOD 00-31

MOD 00-31[162], developed by the MOD in 1987, is a standard relating to the development of safety-critical software for airborne systems. The standard relies on DO-178A [188], a standard developed by the Radio Technical Commission for Aeronautics in 1985, and makes some minor adjustments to that standard. Given the later publication of MOD 00-55 and MOD 00-56, it must be assumed that this standard has been surpassed by the stronger standards which cover procurement and development of safety-critical software for MOD applications wherever the avionics system software is judged safety critical.

The rest of this section deals with RTCA/DO-178A. This standard describes techniques and methods that may be used to develop software for airborne systems. The standard does not specify requirements: it just offers guidance on techniques that help the developer meet the requirements of government regulatory agencies. The standard includes a caveat that with the current state of knowledge of the techniques described in the standard, the recommended techniques may not be sufficient to ensure that safety and reliability targets are achieved, thus, other techniques may need to be employed.

The standard offers guidelines to determine the level of criticality of the software, techniques for software development and configuration management, documentation guidelines, and system design guidelines.

The certification criteria for the system depends on the significance of the functions to flight safety. Determining the criticality category is the first step in determining the certification requirements. There are three categories:

- **Level 1.** Critical, a failure will prevent the safe operation of the aircraft.
- **Level 2.** Essential, a failure reduces the capability of the aircraft or the ability to handle adverse conditions.
- **Level 3.** Non-essential, a failure does not significantly reduce the capability of the aircraft.

The most critical function of the system determines the category of the whole system unless that system has been partitioned into elements of different criticality categories.

The development section helps define a certification plan and software development activities to obtain software at a given criticality category or software level, the latter being analogous to criticality categories. The standard takes the view that the regulatory agencies are primarily interested in the resulting level of safety and not in the way the software was developed.

The description of development techniques follows the "waterfall" model and discusses the documents to be produced, the verification activities, and the assessment activities for each phase of development. No particular techniques are discussed; however, outlines are given which suggest techniques consistent with current, good software development practice.

The standard states that the practices for software configuration management are derived from the current practices for hardware configuration management. It is stated that the software configuration management plan and the software quality assurance plan are closely related and that these two plans may be combined. The plans (separately or combined) address the identification, control, status, accounting, media control, and configuration audits of the operational software and of the hardware and software used to support the development. The intent is to offer visibility of the configuration management process to the installers and regulatory agencies. The configuration management plan centers on the use of part numbers associated with functional components of the system thus defining replaceable units. The plans should identify the disciplines involved in the development, production and post-certification of the project related to configuration management or quality assurance. It is stated that a problem reporting and corrective action procedure should be introduced for Level 1 and Level 2 software and that such a procedure is also desirable for use with Level 3 software.

The standard describes a series of fourteen documents that describe the system. The responsibility for the creation of these documents lies with the development organization. The documents provide information about the software; for example, a programmer's manual, source code in both machine and human readable forms, and executable code are included. Other information in these documents describes the development and support environment, the software requirements, the design description, and management activities such as the configuration management plan or the accomplishment summary, a document used mainly for certification purposes.

5.4 IEC-880

This standard [97], developed for the international nuclear power industry, is primarily concerned with developing software for the safety systems of nuclear power plants. Essentially, it may be considered to consist of two parts. The main body forms the requirements on development and indicates the particular requirements, some rationale, and comments on how the requirements may be met. The second part is a series of appendices giving detailed requirements to back up the requirements offered in the main body.

The standard mandates neither particular techniques nor even classes of techniques. The standard suggests that the project should be divided into a number of self-contained but mutually dependent phases. For any safety-related application, none of the identified phases will be omitted. The entire life cycle must be considered and each phase of the software life cycle should be divided into elementary tasks with a well-defined activity for these tasks. Each product will be systematically checked after each phase. Each phase will end with a critical review (part of the verification process for the project).

The software requirements must be derived from the requirements of the safety systems and describe the product, not the project. They describe what has to be done, not how it has to be done. An appendix offers guidelines for the content of the software requirements. This includes a complete list of system functions with a detailed description that relates the functions to one another and to system inputs and outputs. Risk considerations, recommendations for functions or other safety features, and other items providing background information on specific requirements may be included as they may be background for licensing even if unused in development. The interfaces between the safety system and any other systems will be documented to indicate the specific interfaces and related software requirements. The computer software must continuously supervise both itself and the hardware. This supervision is a primary factor in achieving overall system reliability. The standard notes that the requirements should be presented according to a standard whose formality does not preclude readability.

The requirements should be unequivocal, testable or verifiable, and realizable. The standard suggests that using a formal specification language may help to show the coherence or completeness of the software requirements.

The standard makes some design recommendations, including decomposition of the software into modules, top-down rather than bottom-up development as well as the avoidance of programming tricks, recursive structures and unnecessary code compaction. No particular language is required, although it is recommended that the language should have a thoroughly tested translator and be completely and unambiguously defined. The standard makes some requirements on documentation of the design and the program that includes a software performance specification and other documentation that may assist verification.

Verification is described as addressing the adequacy of the software requirements in fulfilling the safety system requirements assigned to the computer system, of the system software design of fulfilling the requirements, and of the final system source code fulfilling the software performance specification. Verification is to take place according to the verification plan. This plan should be sufficiently detailed so that verification may be performed by an independent group. This group should be managerially distinct from the development group.

The standard makes requirements on the integration phase where software and hardware are put together. The procedures used to put the pieces together depend on the specific project; however, they should be documented in an integration plan and must cover the acquisition of the proper modules, the integration of hardware modules, the correct linkage of software, preliminary tests of the integrated function, and the formal release of the integrated system to verification testing. When the entire computer system is tested; the standard recommends that the tests cover all signal ranges and the ranges of computed or calculated parameters, cover the voting and other logic and logic combinations, be made for all trip or protective signals in the final assembly, and ensure that accuracy and response times are confirmed.

The standard requires that a formal modification procedure should be established that includes verification and validation. Requests to make modification should include the reason for the request, the functional scope, the aim, and the originator. Any request will be evaluated independently resulting in either a rejection, or an approval, or a requirement for further detailed analysis. After a modification has been made, verification and validation must be performed according to the analysis of the impact of the modification.

The standard makes requirements on the operation of the software. The requirements include commissioning tests and man-machine interaction tests to ensure that the operator cannot alter any of the program logic. Further, the operators are required to be trained in the system, including training on a system equivalent to the actual system.

In summary, the IEC-880 standard does not specify particular techniques, but rather discusses the type of work that must be performed in the development of software for the shutdown system of a nuclear power plant. This standard is unusual in the field of standards as it makes requirements about operator training as well as requirements on the software and development process.

5.5 SafeIT

The SafeIT documents ([17], [18]), developed by the UK Department of Trade and Industry, while not a standard in themselves, describe an approach to standards that is technically interesting. SafeIT is described in two volumes; Volume 1 [17] describes the rationale behind the work described in Volume 2 [18].

The aims of the program are to assist in the development of technically sound, feasible, generic international standards with appropriate domain specific standards that are consistent with the generic standards. Secondary aims are to encourage the use of software in safety related applications and to encourage the adoption of best development practices in relation to the software. Related to this is the aim of ensuring that use of software enhances rather than decreases system safety.

The rationale discusses the fact that there are already a number of domain specific standards for the development of safety-related systems. One of the problems is that the domain specific standards are not written in any coherent way, making it difficult to translate solutions from one domain to another. To achieve the aims, Volume 1 identifies four key areas of activity that require a coordinated approach: standards and certification, research and development, technology transfer, and education and training. One of the activities in standards and certification is the development of the standards framework, described in Volume 2.

The second volume details work that has already taken place towards the development of standards. The volume is presented in two parts, the first proposing a framework for safety related standards and the second discussing methods for standards development.

The objectives in developing the standards framework were to develop common concepts and terminology (e.g., concepts such as integrity and risk), to develop a set of agreed-upon principles, to develop an agreed-upon set of safety objectives for the assurance of integrity in software systems (the safety objectives should be common to all applications and levels of safety), to provide information about technical and process oriented techniques, to develop a method that can be employed by standards groups so that they can systematically develop standards that meet the safety objectives, to give examples of requirements for each level of safety, and to allow existing and proposed standards to be incorporated into the framework.

The structure of the framework will be based on core standards that relate to the framework. Surrounding these core standards will be generic and domain-specific standards. Auxiliary standards that define activities or techniques or methods will support the generic and domain-specific standards.

Auxiliary standards defining particular activities, techniques, or methods can be separated out and developed to support most domains. SafeIT has already identified candidate activities for auxiliary standards such as quality assurance, configuration management, programming languages, hazard analysis techniques, and others. The use of auxiliary standards means that the framework developers should be able to take advantage of existing standards work. The fundamental concern when developing an auxiliary standard is to consider whether it addresses an aspect of software development that can be separated usefully. Examples are topics not specific to safety such as security or reliability, or a self-contained technique or method.

The core standards in the proposed framework will include standards describing common safety principles, definitions of common terms and concepts and a common standards development method. The method is necessary if there is to be a possibility of harmonizing the various standards. The second part of Volume 2 describes the core standards in greater detail.

A number of common principles are outlined These include principles such as safety being a system rather than a software concern, that safety must be built into a system rather than added on, and that the acceptable level of safety is a balance between the risks, benefits and costs.

The discussion of terms and concepts centers around the fact that there are already a number of standards activities with differing views on many fundamental concepts such as the number of levels of safety within a system. For the framework approach to work, it must describe the most general set of concepts onto which the concepts employed in existing standards may be mapped.

The life-cycle concept is important since it is in the context of a life-cycle that terms and concepts have meaning. The framework proposes to focus on three different types of life-cycles: safety, procurement, and development. The approach taken has been to consider each life-cycle as a group of sufficiently large and general phases into which real models can be fitted. The document makes it clear that although specific proposed standards have been adopted for the framework, that this adoption is not intended to limit discussion or use of other life-cycles, but rather as a focus for discussion.

Certain roles across all of the standards are also expected to be identified and standardized. These roles include procurers, developers, users, and others. Terms that are used to describe the framework itself are also expected to be standardized.

A standard contains requirements on the process or product. The method proposed for developing standards discusses what factors should be taken into consideration for the development of standards. These factors start with the definition of the overall objectives of the standard. These overall objectives are refined into a set of more detailed objectives and a range of techniques that can meet these objectives is outlined. Integrated sets of techniques are selected and the rationale for the selection is documented. Finally, the standard is produced describing each the objectives, the techniques expected to satisfy the objectives and the rationale for the choice of techniques. It is noted that some techniques exclude others while others fit particularly well. Standards writers are warned to be aware of the problems of combining techniques. Additionally, they are advised that schemes should be developed that reduce the possibility of the users of the standards selecting inappropriate combinations. The method outlined above is then used to further define the framework standard.

In summary, the SafeIT approach is the development of a framework into which particular standards may be incorporated. The framework includes a method for developing standards that will fit into SafeIT and will, the authors claim, lead to clearer standards for the developers of safety-critical software. The SafeIT approach is interesting as it proposes an approach that will allow for the potential unification of the conflicting and competing maze of international standards.

5.6 Effects of Standards

Standards may have a number of positive effects including the provision for a common architecture, a common vocabulary, and a statement of a minimal level of compliance from the community. They may also, however, have some negative effects. These effects are discussed below.

5.6.1 Standard Is Inappropriate

A standard may be inappropriate for a number of reasons. The most likely reasons are that either the standard is outdated or the technology defined in the standard may not be readily applicable.

A standard may be out of date because it takes a long time to create or revise standards. As the process proceeds the standard takes on an inertia with respect to change and becomes more resistant to changes based on current technologies. Once released and accepted by the community, there is community resistance to changes in the standard because the community may have to change their practices to be compliant with the new standard. Thus, standards have long life spans and, in a rapidly changing technological area such as software engineering, are often out of date (sometimes even before the standard is released). To prolong the useful life span of a standard, the standards developers may attempt to standardize on a technology that is only just coming out of the research community. Generally, the standardization committee has a belief in the value of such research technology and may even perform a number of experiments to convince itself of the value of the technology. Unfortunately, the technologies may not be proven to operate on large-scale systems or in a domain as wide as the domain to which the standard applies. Further, the technology may simply be inappropriate for some systems for which the standard requires its use.

5.6.2 Standard Is Ineffective

If the purpose of a standard is to bring the community to an acceptable level of quality, it is important that the level of quality defined by the standard is acceptable to the system procurers.

The contents of a standard depend in large part on the members of the committee given the task of the creation of the standard. If the committee is made up largely of developers then it is in the best interests of the committee members to make the standard as weak as possible so that they will not have to change their current practices to conform to the standard. If, on the other hand, the committee is made up largely of procurers, the standard may indeed have sufficient strength to bring the development practices of the community up to the desired level. Unfortunately, there are many standards which have been constructed by the development rather than procurement communities. This practice has led to a large number of ineffective standards.

5.6.3 Standard Induces Minimal Compliance

Standards describe a minimum level of compliance, and the minimum practices and technologies that developers must employ for the development product to be acceptable. Unfortunately, there is a distinct possibility that a developer may look at the standard as the maximum level of quality to which they need to aspire. After all, once the standardized practices and technologies have been employed, the developers have met the criteria. Why do more, especially if doing more than required by the standard costs money? Thus, the standard may induce minimum levels of compliance rather than being the hoped-for minimum.

Related to the issue of minimum compliance is an interesting issue of liability. The question, currently unresolved, is "Who is liable if a product developed according to a standard fails?" The developers will argue that they met the standard and thus were not negligent in the development of the product.

6 Conclusions

Finally, we present the conclusions drawn through the creation of the annotations in the bibliography and the writing of this report. We then describe potential future work in the development of safety-critical systems.

6.1 Conclusions

The creation of software to be included in safety-critical systems is a hard task. Certainly it requires more care and thought than the development of software in other systems. Whenever developers are creating software that may, within the context of a system, threaten life or property, they must use the most careful approaches possible. In many cases, it will not be possible to adequately test the software in an operational situation because these cases involve systems that must not be allowed to fail—weapon systems are one such class of systems.

Safety is not an attribute that can be added to software after the event; it must be designed into the software from the start, and it must be constantly checked to ensure that unexpected, unsafe, functions have not been added or necessary functions have not been removed. Thus, development of safety-critical software depends on appropriate system requirements engineering, system hazards identification and system design and software requirements engineering, design and development.

System engineering is particularly important because we still have an imperfect understanding of the ways in which software failures can affect the system. It is important, wherever possible, to offer alternative backups to the safety-critical software that allow the system operators to perform degraded, yet safe, operation of the system.

There is no substitute for high-quality developers, particularly when determining the ways in which the system may fail and thus lead to potential mishaps.

Software fault tree analysis is a promising approach to ensuring that the software will not lead to a mishap; however, it relies on knowing in advance what the possible failures of the system could be. The use of fault tree analysis for analysis of the design and the specification is a valuable step towards ensuring that the system will be safe early rather than late in the development process. Unless there is improvement in the determination of conformance of an implementation to a design or specification, however, the use of fault tree analysis at the specification or design levels will not replace the use of safety analysis at the implementation level.

Formal methods are discussed throughout the literature as a potential solution to the issue of ensuring conforming implementations. However, these are not a complete solution. While the use of formal methods helps reduce system faults by not inserting errors into the implementation, they do not help with issues of random, low-probability faults such as a component failure in the computer or sensors. These faults need to be masked using fault-tolerant techniques both in the system hardware and software. Further, formal methods, by themselves, do not remove errors introduced at the level of the system requirements.

A requirements (or specification) error may lead to implementations that conform to the written description of the system but still result in an unsafe system. Thus the requirements and specifications must be analyzed to ensure that an acceptable level of risk has been achieved. Of particular importance are the tradeoffs between safety and other quality factors such as performance or security.

Standards can help in the development of safety-critical systems in that they state guidelines by which the system should be developed. However, they are not a panacea as they may be out of date, inapplicable to the particular domain, or induce only minimal compliance. It should be stressed that standards describe the minimum effort required to engineer safe systems and that developers should be strongly encouraged to exceed the standards.

6.2 Further Work

This report has been an attempt to capture in written form the information obtained from reading the literature on software safety. There aro many directions which may be pursued from this point.

1. There are various classes of safety-critical systems in operation. Examples of different classes are nuclear reactor shutdown systems which, after detecting a hazardous state, perform hazard recovery by shutting down the reactor and then need not operate again and an avionics flight control system which, after detecting a hazardous state must avoid the hazard and then continue to operate and control the aircraft. These different classes may work best with different architectural designs.

   A valuable contribution to the development of safety-critical software would be a classification of criticalities with indications of architectures that have been accepted as safe in the different classifications. Then, for any new system, the developers could classify their system and use that classification to suggest a requirements model and an architectural structure for the system.

2. Convincing examples of the application of the techniques described in this report are needed. The inertia found in development organizations is such that even though the problem of developing safe software will not go away, developers still need to be convinced that more formal techniques will help solve the development problem. A number of techniques have been developed and tested by academics on small examples (there have also been some publications concerning significant systems such as the Darlington nuclear reactor [177]). Examples of significant application of the techniques described in the paper a,e one way to convince developers.

3. There are currently many standards pertaining to software safety with varying levels of safety induced through the use of the standards. One problem is that many of the existing standards are out of date. One of the more interesting approaches to standards is the notion of a standards framework into which

standards specific to a domain may be inserted. Consistent with this view is the notion of a meta-standard which describes the nature of the techniques that are to be employed and allows the developers to instantiate the standard with techniques currently being used by the developers. Such an approach means that the standards body do not standardize out-of-date or untested technology. The disadvantage is that the people responsible for deployment of the system must first approve the development approach recommended by the system developers. This requires new skills for those responsible for system deployment.

4. There appear to be two almost competing camps in the development of safety-critical software. The formalists who suggest that errors (and therefore hazards) can be eliminated by using formal methods; the safety-analysts examine the system artifacts after construction for potentially hazardous behaviors. It seems reasonable that a combination of the two approaches will lead to the development of the safest software and that the interactions between these two approaches should be investigated further.

Annotated Bibliography

[1] H. Abbott. *Safer by Design: The Management of Product Design Risks Under Strict Liability.* The Design Council, London, 1987.

The book comprises three parts: a brief overview of the legal background on product liability in Europe and the United States, an overview of approaches used to manage product risks introduced in the design phase, and a number of case histories detailing failures in design. A product is defined as defective when "it does not provide the safety which a person is entitled to expect, taking all circumstances into account." Management of risks uses a four part strategy: identification of risks, a risk reduction program, a risk transfer program (insurance or contracts limiting liability), and a risk retention program. Risk avoidance is considered to be part of risk reduction. The book devotes chapters to each of the parts of the strategy. The final part of the book is a selection of case studies of various accidents. The Amoco Cadiz accident includes a number of fault trees used to indicate alternative, safer designs.

[2] R.J. Abbott. Resourceful Systems for Fault Tolerance, Reliability, and Safety. *ACM Computing Surveys*, 22(1):35-68, March 1990.

[3] Air Force Inspection and Safety Center. *Software System Safety*, AFISC SSH 1-1 edition, September 1985.

[4] Air Force Inspection and Safety Center. *Software System Safety Handbook.* AFISC SSH, Norton Air Force Base, CA, 1985.

[5] T. Anderson, editor. *Safe and Secure Computing Systems.* Blackwell Scientific Publications, 1989.

Collected proceedings of the 1986 Safety and Security Symposium held by the Centre for Software Reliability. The aim of the symposium was to consider techniques applied to safety, security, and dependability and see how these techniques applied to other areas. For example, the security notion of a separation kernel may have value for safety applications. Papers from this symposium of relevance to software safety are listed separately.

[6] J. Arlat and K. Kanoun. Modelling and Dependability Evaluation of Safety Systems in Control and Monitoring Applications. In *IFAC SAFECOMP '86*, pages 157-164, Sarlat, France, October 1986.

[7] J. Arlat and J.C. Laprie. On the Dependability Evaluation of High Safety Systems. In *15th International Symposium on Fault Tolerant Computing*, pages 318-323. IEEE Computer Society Press, June 1985.

[8] W.B. Askren and J.M. Howard. Lessons Learned from Computer Aided Industrial Machine Accidents. In *COMPASS '87 Computer Assurance*, Washington, D.C., July 1987.

[9] J.B.J. van Baal. Hardware/Software FMEA Applied to Airplane Safety. In *Annual Reliability and Maintainability Symposium*, pages 250-255, Philadelphia, PA, January 1985.

[10] A. Ball and R.N..H. McMillan. A Review of Development in Creating and Proving Reliable Software. In *American Nuclear Society International Topical Meeting on Computer Applications for Nuclear Power Plant Operation and Control*, pages 254-260, Richland, WA, September 1985.

[11] L. Bass. Cost-effective Software Safety Analysis. In *Annual Reliability and Maintainability Symposium*, pages 35-40, Atlanta, GA, January 1989.

[12] L. Bass. Products Liability: Design and Manufacturing Defects. Shepard's/ McGraw-Hill, 1989.

This book is a legal reference work intended primarily for people involved in litigation over issues of product liability. Thus, much of the material, although fascinating, is not relevant to software safety. Chapter 7 discusses issues of system safety engineering, and the September 1990 supplement adds considerable detail concerning issues of software safety. Other chapters have some relevance. For example, warnings that a system may be hazardous are not a substitute for a redesign that would reduce the probability of or eliminate a hazard occurrence.

[13] L. Bass and C. Hoes. System Safety Analysis of Software-controlled Robotic Devices. *Industrial Management*, 29(1):17-21, Jan-Feb 1987.

[14] H. Bassen, J. Sillberber, F. Houston, W. Knight, C. Christman, and M. Greberman. Computerized Medical Devices: Usage trends, Problems and Safety Technology. In *Seventh Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 180-185, Chicago, Ill., 1989.

[15] M.H. Bell. Software Safety Analysis. In *COMPCON '84: The Small Computer (R)Evolution*, pages 353-363, Arlington, VA, 1984.

The paper presents a corporate approach to safety analysis. The approach uses three phases. The first is analysis of the specification. System safety requirements are derived using guidelines in appropriate standards. These requirements are translated into specification designs that conform to certain minimum standards. The specification is analyzed using a specification flow diagram for a number of desirable properties. Software hazard analysis is performed at the specification level. This uses inputs from a fault tree analysis (when available from a software fault tree analysis). A final result of the specification analysis phase is the establishment of test requirements. The second phase is analysis of the software design and coding. Software design analysis concentrates on the documentation, design, mathematics, algorithms, input device, and overlapping conditions to determine if the code agrees with the safety requirements and specification. Software coding analysis uses an emulator to execute the compiled code to ensure compliance with various inspection crite-

ria. The results of the coding analysis are summarized using a software subsystem hazard analysis which is initially based on the design analysis and updated according to the code analysis. The third phase of the safety analysis is a final report summarizing the work done in the first two phases. The paper claims that using these techniques will lead to cost savings and reduced implementation time and will lead to credible, enhanced software safety.

[16] P. Bennett. Safety Critical Control Software. *Control and Instrumentation*, 20(9):75-77, September 1988.

[17] R.E. Bloomfield. *SafeIT—The Safety of Programmable Electronic Systems.* Department of Trade and Industry, London, UK, June 1990.

This document lays out the rationale and aims of the SafeIT project. The intent is to create a standards framework into which specific safety standards may be fitted. The reason for doing this is the growing maze of standards relating to safety that compete rather than cooperate with each other.

[18] R.E. Bloomfield and J. Brazendale. *SafeIT—A Framework for Safety Standards.* Department of Trade and Industry, London, UK, May 1990.

This is a companion to the SafeIT overview [17]. This document describes a proposed framework for standards. Essentially, the framework consists of core standards defining common terms, concepts and principles, with generic and domain specific safety standards. There is also the notion of an auxiliary safety standard which is a stand-alone standard of general relevance. For example, a standard on the application of a particular method would be an auxiliary standard.

[19] R.E. Bloomfield and P.K.D. Froome. The Application of Formal Methods to the Assessment of High Integrity Software. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986.

[20] Boeing Aerospace & Electronic Systems. *Safety Engineering & Management, BA&E System Safety Instruction - System Safety Engineering in Software Development (Draft)*, November 1989.

[21] S. Bolobna and D.M. Rao. Testing Strategies and Testing Environment for Reactor Safety System Software. In *IFAC SAFECOMP '86*, pages 179-184, Sarlat, France, 1986.

[22] S.E. Bologna et al. An Experiment in Design and Validation of Software for a Reactor Protection System. In *Proc. SAFECOMP '79*, pages 103-115, 1979. [23]

[23] B.J. Bonnett. Software Assurance for Safety and Security Critical Systems. In *COMPCON '84: The Small Computer (R)Evolution*, page 352, Arlington, VA, 1984.

Outlines the work of the Software System Safety Working Group who have identified four areas of study for software safety.

1. *Programmatic awareness—making system managers aware of the risks that software will introduce into their systems.*
2. *The generation of accurate and effective safety design requirements in the initial system and subsystem specifications.*
3. *Procedures to ensure that software design does in fact design the safety features into the code.*
4. *Analysis techniques that can effectively (and cheaply) review the final code to verify system safety.*

[24] B.J. Bonnett. Software System Safety. In *Seventh Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 186-192, Chicago, Ill, 1985.

[25] W.C. Bowman et al. An Application of Fault-Tree Analysis to Safety-Critical Software at Ontario Hydro. In *Probabilistic Safety Assessment and Management*, G. Apostolakis, editor, pages 363-368, New York, 1991. Elsevier.

[26] J.D. Bronzino, E.J. Flannery, and M. Wade. Legal and Ethical Issues in the Regulation and Development of Engineering Achievements in Medical Technology, Part I. *IEEE Engineering in Medicine and Biology Magazine*, pages 79-81, March 1990.

[27] J.D. Bronzino, E.J. Flannery, and M. Wade. Legal and Ethical Issues in the Regulation and Development of Engineering Achievements in Medical Technology, Part II. *IEEE Engineering in Medicine and Biology Magazine*, pages 53-57, June 1990.

[28] M.J.D. Brown. Rationale for the Development of the UK Defence Standards for Safety-Critical Computer Software. In *Fifth Annual Conference on Computer Assurance*, pages 144-150, Gaithersburg, MD, June 1990.

[29] M.L. Brown. Software Safety for Complex Systems. In Seventh Annual Conference of the *IEEE Engineering in Medicine and Biology Society*, pages 210-216, Chicago, Ill., 1985.

[30] M.L. Brown. What is Software Safety and Whose Fault Is it Anyway? In *COMPASS '87 Computer Assurance*, pages 70-71, Washington, D.C., July 1987.

Argues that safety is a system issue and that the software must be viewed as a subsystem that must be fully integrated into the system safety program. The responsibility for safety is primarily the system safety engineer's, however, the software developers must also accept some responsibility. The software developers must develop the software functions within the context of the system (not an approach generally supported by current military standards). The conclusion is that software systems safety is complex and involves all aspects of development and that it requires a strong commitment to system engineering and system safety.

[31] M.L. Brown. Software Systems Safety and Human Errors. In COMPASS '88 Computer Assurance, pages 19-28, Gaithersburg, MD, July 1988.

Makes the point that some software used in the development of a safety-critical system is in itself safety-critical. The paper briefly describes the tasks of MIL-STD-882B relating to software safety. The use of a requirements traceability matrix for safety-critical software is described, though it is left to the reader to determine the efficacy of this approach. The paper divides human error into four classes; design, coding and testing, operational use, and software maintenance errors. For each class of error, discussion of the techniques intended to minimize the likelihood of error occurring is included. The paper discusses the interactions between human factors issues and safety issues, particularly with respect to the user interface analysis.

[32] W. Bryan and S. Siegel. Software Product Assurance—Reducing Software Risk in Critical Systems. In *COMPASS '88 Computer Assurance*, pages 67-74, Gaithersburg, MD, July 1988.

The paper argues that product assurance improves the reliability of critical systems by means of increasing visibility into the development process. Product assurance is defined as verification and validation, test and evaluation, configuration management, and quality assurance. The paper continues with some examples that estimate how product assurance could detect potential problems. The paper concludes with the argument that additional initial expense in development by using product assurance techniques will save much greater expense later on in either the development or operation of the system.

[33] B.A. Burton and R.L. Rathgeber. An Integrated Approach to Software Reliability. In *Seventh Annual Hawaii International Conference on System Sciences*, pages 3-12, 1984.

[34] J.E. Cantu and C.R. Turner. B-1B Approach for Test Coverage of Safety Critical Software. In *COMPASS '87 Computer Assurance*, pages 56-62, Washington, D.C., July 1987.

The paper asserts that the best way to prove software correct is through testing. The paper describes the three phases of testing performed on theB1-B terrain-following software: module test and integration, software system test, and system validation test. The testing includes a software path instrumentation system which, for any given test, determines which lines of code were executed

and which lines of code should have been executed. If there is a discrepancy, then either there is an error in the code or the test is not sufficiently thorough.

[35] D.N. Card and D.J. Schultz. Implementing a Software Safety Program. In *COMPASS '87 Computer Assurance*, pages 6-12, Washington, D.C., July 1987.

Mentions that techniques applicable to software safety have been borrowed from hardware and that concern for safety must permeate the life cycle for critical systems. The paper describes experimental results which suggest that independent verification and validation may not reduce the total number of errors in a system, but do add to the cost. They note that IV&V has been successful in identifying specific classes of errors when tasked so to do. The main content of the paper discusses management issues with respect to organization developing software for safety-critical systems.

[36] D.B. Cazden. Software Sneak Analysis as a Development Support Tool. In *Seventh International Software Safety Conference*, pages 2.6-2-1—2.6-2-5, 1985.

[37] C. Cha, N.G. Leveson, and T.J. Shimeall. *Safety Verification of Ada Programs in Murphy.* Technical Report TR 87-23, Department of Information and Computer Science, University of California, Irvine, 1987.

This report is essentially the same as the IEEE paper [136] by the same authors. The report is more detailed in its initial discussion in that it suggests that one option toward safety-critical systems is that they should not be built using software. The report also discusses the assumptions under which backward analysis can be successfully performed: 1) that not all failures are of equal consequence, and 2) that only a relatively small number of failures are potentially serious. The report presents the same fault tree templates for Ada statements and also uses the traffic light example as used in the IEEE paper. The report discusses some of the Murphy tools used to support fault tree analysis. At the time the report was written, a fault tree editor and a fault tree artist (a display tool) existed and a fault tree generator was under construction. This latter tool takes an Ada program as input and, in conjunction with the analyst, produces a fault tree as its result. The report concludes by stating that the approach is human-oriented and that although tools can help, they cannot substitute for the skill of the analyst.

[38] S.S. Cha. *A Safety-Critical Software Design and Verification Technique.* Ph.D. dissertation, ICS Dept., University of California, Irvine, 1991.

[39] P.C. Clements. Engineering more secure software systems. In *COMPASS '87 Computer Assurance*, pages 79-81, Washington, D.C., July 1987.

Considers safety, security, and other requirements as special cases of a general desire to ensure that a particular hardware and software system behaves as expected. In order to ensure that this is the case, the first step is to write down what is expected of the system. The A7-E style of writing specifications

of requirements is briefly discussed with some arguments that indicate it is possible to be confident that an implementation satisfies the specification.

[40] U.S. Atomic Energy Commission. Reactor Safety Study: An Assessment of Accident Risks in the U.S. Commercial Nuclear Power Plants. Report WASH-1400, 1975.

[41] Committee on Science, Space, and Technology, U.S. House of Representatives. *Bugs in the Program—Problems in Federal Government Computer Software Development and Regulation—Staff Study*. Technical report, U.S. Government Printing Office, Washington, D.C., September 1989.

[42] B. Connolly. Software Safety Goal Verification using Fault Tree Techniques. In *COMPASS '89: IEEE Fourth Annual Conference on Computer Assurance*, pages 18-21, Gaithersburg, MD, 1989.

[43] B. Connolly. Software Safety Goal Verification using Fault Tree Techniques: A Critically Ill Patient Monitor Example. In *Second Annual IEEE Symposium on Computer-Based Medical Systems*, pages 118-120, Minneapolis, MN, June 1989.

[44] S.D. Crocker. Techniques for Assuring Safety—Lessons from Computer Security. In *COMPASS '87 Computer Assurance*, pages 67-69, Washington, D.C., July 1987.

The paper uses the history of the development of security-critical systems as a predictor of the future development of safety-critical software. The potentially overlapping phases are described as heightened visibility for such systems, introduction of new methodologies, increased availability of hardware support, use of formal specification, and introduction of formal tools. The paper makes the point that a key to building safer systems is an increase of effort on requirements for safety. The paper concludes that any and all of the approaches are necessary to ensure increased safety.

[45] W.J. Cullyer and W. Wong. A Formal Approach to Railway Signalling. In *COMPASS 90: Computer Assurance*, Gaithersburg, MD, July 1990.

[46] N.C. Dalkey. *The Delphi Method. An Experimental Study of Group Opinion.* RM 58-88 PR, The Rand Corporation, 1969.

[47] B.K. Daniels, editor. *Achieving Safety and Reliability with Computer Systems.* Elsevier Applied Scientific, November 1987.

The proceedings of a conference held in 1987 by the Safety and Reliability Society. The structure of the book follows that of the symposium. Identified trends are the increased use of formal methods in industry with increased tool support. Complementing formal methods is the continuing best use of accumulated experience with software engineering and safety and reliability assessment. Papers of relevance are annotated separately.

[48] B.K. Daniels, R. Bell, and R.I. Wright. Safety Integrity Assessment of Programmable Electronic Systems. In *Proc. SAFECOMP '83*, pages 1-12, 1983.

[49] H.T. Daughtrey. Experiences in Conducting Independent Verification and Validation of Safety Parameter Display System Software. In *American Nuclear Society International Topical Meeting on Computer Applications for Nuclear Power Plant Operation and Control*, pages 267-273, Richland, WA, September 1985.

[50] R. De Santo. A Methodology for Analyzing Avionics Software Safety. In *COMPASS '88 Computer Assurance*, pages 113-118, Gaithersburg, MD, July 1988.

Describes an approach to a method that helps identify safety-critical software functions and helps isolate the safety critical paths. The method is driven by documents available if the development is using the 2167-A standard. The method helps the analyst gain a greater understanding of the system by leading the analyst through the system in a structured manner. Safety critical hardware signals are used as the primary source for identifying the operationally related safety-critical software function.

[51] E.S. Dean Jr. Software System Safety. In *Proc. Fifth International System Safety Conference*, 1981.

[52] D.E. Denning. Secure Databases and Safety. In T. Anderson, editor, *Safe and Secure Computing Systems*, pages 101-111. Blackwell Scientific Publications, 1989.

The paper discusses the four categories of security requirements on database systems: authorization, data consistency, availability, and identification, authentication, and audit. The applicability of the security policies to system safety is shown in each category. The paper concludes by showing how database security policies assist in the generation of a safe system. However, the paper also point out ways in which the security policy may conflict with the safety policy and tentatively suggests ways in which these conflicts may be avoided.

[53] Department of Defense. *Military Standard 1629A: Procedures for Performing a Failure Mode, Effect and Criticality Analysis.* Department of Defense, 1984.

This standard describes failure modes, effects and criticality analysis and the circumstances under which such analysis should be applied. The standard does not apply to software, but to hardware to be acquired by the DoD.

[54] Department of Defense. *Military Standard 882B: System Safety Program Requirements.* Department of Defense, 1984.

This standard outlines the tasks that must be described in the program contract to satisfy DoD regulations on system safety. The task areas are divided into *three areas: those associated with management, those with system design, and those with software. For each area, the tasks are described in terms of what the task must achieve. The standard does not require specific techniques, but requires that the program manager either choose techniques or determine whether contractor proposed techniques will lead to an acceptable level of system safety.*

[55] M.S. Deutsch and R.R. Willis. *Software Quality Engineering: A Total Technical and Management Approach.* Prentice Hall, 1988.

The book covers managerial aspects of inducing quality into software in greater detail than it covers technical aspects. Safety is considered as one of the twenty seven criteria that make up quality and a few suggestions are made which are claimed to help with safety management. Some techniques such as software fault tree analysis, reliability modeling, multi-version software, and correctness proofs are outlined as approaches for achieving exceptional quality.

[56] J.H. Dobbins. Software Safety Management. In *COMPASS '88 Computer Assurance*, pages 108-112, Gaithersburg, MD, July 1988.

The paper focuses on an approach for life cycle management of software safety continuing into operational phases. Discusses the current government practice of writing requirements in prose and describes it as the most error prone way to describe requirements. There is discussion of other approaches to describing requirements, such as PSL/PSA and data flow diagrams. Fagan inspections of the design and code are recommended as a way of reducing 70% of defects prior to unit testing. Automated support for analyzing code is also discussed. The use of call path analysis is discussed, any path which includes a safety-critical module or an overly complex module is marked for exhaustive analysis and stress testing. The results of the call path analysis are used for determining tests that ensure 100% coverage of the system. The paper concludes with remarks that safety management must be carried out throughout the development process and on into the maintenance phase.

[57] E.L. Duke. V & V of Flight and Mission-Critical Software. *IEEE Software*, 6(3):39-45, May 1989.

Discusses a verification and validation method used at NASA Ames-Dryden. Analysis and testing are performed on abstract models of the system. The models include linear-system models, aggregate-system models, block diagrams, schematics, specifications, and simulations. They prototype flight software which is evaluated by pilots and engineers; the prototype is then used as the basis for a specification from which the actual flight software is produced. Testing takes place by providing the real software identical input to the prototype and comparing results. There is brief discussion of limitations of the Ames-Dryden approach and that formal proof and statistical analysis address the chief limitations. They have three levels of criticality, from failure causing loss of life or limbs or damage to public safety to systems whose failure may produce inaccurate results or inefficient use of resources. More effort is placed into the higher levels of criticality.

[58] J.R. Dunham. Measuring software safety. In *COMPCON '84: The Small Computer (R)Evolution*, pages 192-193, Arlington, VA, September 1984.

Presents an approach to measuring software safety using repetitive run analysis. The paper accepts that this is one form of testing to measure safety and that other approaches exist. It also makes the distinction between measuring reliability and safety; the latter must not only estimate the frequency of errors, but also their severity.

[59] J.R. Dunham. V & V in the Next Decade. *IEEE Software*, 6(3):47-53, May 1989.

Discusses some factors affecting verification and validation such as age of the software, reuse, and criticality. Predicts that in the next decade V&V technology will be mature, covering all phases of the life cycle, that it will be included in software development environments, that it will rely on formal verification and statistical quality control, and that it will have guidelines that help select and combine techniques. Mentions some uses of formal verification techniques.

[60] M. Dunn and W. Hillison. The Delphi Technique. In *Cost and Management*, pages 32-36. 1980.

[61] L.G. Egan. *Analysis of the Certification Process of Computer Programs Used in a Nuclear Power Plant, Using the Management Systems Approach.* Technical report, Software Certification Institute, Santa Maria, Ca., Year unknown.

[62] W.D. Ehrenberger. Fail-Safe Software—Some Principles and a Case Study. In B.K. Daniels, editor, *Achieving Safety and Reliability with Computer Systems*, pages 76-88, September 1987.

The paper argues that one way to achieve safety is by having the software follow previously executed paths and, whenever a new path is discovered, take some system-specific safety action. The information on previously executed paths is generated during testing and may either be control-flow oriented or data flow oriented. Control-flow monitoring is achieved by building a tree of basic block entries during testing such that each path from root to leaf is a trace of an execution. Data-flow monitoring performs a similar task, but uses array addressing points rather than basic block as the data from which the tree is built. During execution, the trace may be compared against the tree for validity. Limitations of the approach are that test data must be similar to operational data, that timing problems and numerical calculation errors are not handled, an that there is considerable execution overhead, both in space and time requirements.

[63] C.A. Ericson Jr. Software and System Safety. In *Proc. Fifth International System Safety Conference*, vol. 1, part 1, pages III-B-1 to III -B-11, Denver, 1981.

[64] EWICS TC 7. *Guidelines for the Maintenance and Modification of Safety-Related Computer Systems.* EWICS, November 1987.

[65] EWICS TC 7. *Safety Assessment and Design of Industrial Computer Systems: Techniques Directory.* EWICS, November 1987.

[66] Food and Drug Administration. *Reviewer Guidance for Computer-Controlled Medical Devices (Draft).* Food and Drug Administration, July 1988.

[67] H.H. Frey. Safety and Reliability—Their Terms and Models of Complex Systems. In *Proc. SAFECOMP '79*, pages 3-10, 1979.

[68] A.W. Friend. An Introduction to Software Safety. In *Seventh Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1232-1237, Chicago, Ill., 1985.

[69] R.C. Fries and R.T. Riddle. A Software Quality Assurance Procedure to Assure a Reliable Software Device. In *Second Annual IEEE Symposium on Computer Based Medical Systems*, pages 135-138, Minneapolis, MN, June 1989.

[70] P. Froome and B. Monahan. The Role of Mathematically Formal Methods in the Development and Assessment of Safety Critical Systems. *Microprocessors and Microsystems*, 12(10):539-546, December 1988.

[71] R.U. Fujii. Software Safety Analysis Is an Integral Part of Systems Engineering, Not a Separate Adjunct. In *COMPASS '87 Computer Assurance*, page 73, Washington, D.C., July 1987.

Argues that software safety analysis must be part of the system engineering process. The most important factor being that during concept and design formulation tradeoffs between performance and safety must be considered to achieve optimal system features.

[72] K. Geary. Beyond Good Practices—A Standard for Safety Critical Software. In B.K. Daniels, editor, *Achieving Safety and Reliability with Computer Systems*, pages 232-241, September 1987.

Describes the changes to the UK Naval Engineering Standard 620 (NES 620) made for safety-critical software. The paper is interesting in that it provides reasoning similar to that behind the development of MOD 00-55 and MOD 00-56. Indeed, the author argues that the changes written into the standard are sufficiently similar to those of the MOD standards and that NES 620 should be superseded by the MOD standards.

[73] G. Gloe and O. Nordland. Qualification and Licensing of Computer-Based Systems for Safety Tasks in German Light Water Reactors. In *American Nuclear Society International Topical Meeting on Computer Applications for Nuclear Power Plant Operation and Control*, pages 326-329, Richland, WA, September 1985.

[74] G. Gloe and G. Rabe. Experience with Computer Assessment. In *Safety and Reliability of Programmable Electronic Systems*, pages 145-151, Essex, England, 1986. Elsevier.

[75] S.G. Godoy and G.J. Engels. Software Sneak Analysis. *American Institute of Aeronautics and Astronautics* (77-1386), pages 63-67, 1977.

[76] J. Goldberg. Some Principles and Techniques for Designing Safe Systems. *ACM SIGSOFT Software Engineering Notes*, 12(3):17-19, July 1987.

[77] D.I. Good. Predicting Computer Behavior. In *COMPASS '88 Computer Assurance*, pages 75-83, Gaithersburg, MD, July 1988.

The paper argues that the only way to assure the safety of a software system is to be able to predict that the system will behave acceptably in the future. A mathematical model of a computer system is described and used to derive equations which enable arguments on the number of states, requirements on acceptance tests, and the effectiveness of testing in general to be derived. The paper discusses issues of completeness, magnitude, instability, the definition of acceptable behavior, and approaches to the demonstration of acceptable behavior. It is argued that the only way to scale up to real systems is by use of all available mathematics and an approach is described. The paper concludes with a comparison of a vision of future practice where prediction of the computer system is possible against current practice where very little of the necessary mathematical foundation exists.

[78] J. Gorski. Design for Safety Using Temporal Logic. In *IFAC SAFECOMP '86*, pages 149-155, Sarlat, France, 1986.

[79] J. Gorski. Formal Support for Development of Safety Related Systems. In B.K. Daniels, editor, *Achieving Safety and Reliability with Computer Systems*, pages 14-28, September 1987.

[80] R. Greenberg. Software Safety Using FTA Techniques. In *Safety and Reliability of Programmable Electronic Systems*, pages 86-95, Essex, England, 1986. Elsevier.

[81] J.G. Griggs. A method of software safety analysis. In *Proc. 5th Int. System Safety Conf.*, volume 1, part 1, pages III-D-1 to III-D-18, Denver, 1981.

[82] G. Gruman. Software Safety Focus of New British Standard, Def Stan 00-55. *IEEE Software*, 6(3):95-97, May 1989.

[83] G. Guiho and C. Hennebert. Sacem Software Validation. In *12th International Conference on Software Engineering*, pages 186-191, Nice, France, March 1990.

[84] M.D. Hansen. Survey of Available Software-Safety Analysis Techniques. In *Annual Reliability and Maintainability Symposium*, pages 46-49, Atlanta, Ga., January 1989.

[85] M.D. Hansen and R.L. Watts. Software System Safety and Reliability. In *Annual Reliability and Maintainability Symposium*, pages 214-217, Los Angeles, Ca., 1988.

[86] L. Hatchard. Applying the Principles of the HSE Guidelines to Programmable Electronic Systems in Safety-Related Applications. *Safety & Reliability*, 8(1):30-36, spring 1988.

[87] D.L. Hauptmann. A Systems Approach to Software Safety Analysis. In *Proc. Fifth International System Safety Conference*, 1981.

[88] K. Hayman. An Analysis of Ordnance Software Using the Malpas Tools. In *Fifth Annual Conference on Computer Assurance*, pages 86-94, Gaithersburg, MD, June 1990.

[89] Health and Safety Executive. Programmable Electronic Systems in *Safety-Related Applications*, Volume 1, An Introductory Guide. Her Majesty's Stationery Office, London, England, 1987.

[90] Health and Safety Executive. Programmable Electronic Systems in *Safety-Related Applications*, Volume 2, General Technical Guidelines. Her Majesty's Stationery Office, London, England, 1987.

[91] K.A. Helps. Some Verification Tools and Methods for Airborne Safety-Critical Software. *Software Engineering Journal*, pages 248-253, November 1986.

[92] M.F. Houston. What Do the Simple Folk Do? Software Safety in the Cottage Industry. In *COMPASS '87 Computer Assurance*, pages S-20—S-24, Washington, D.C., July 1987.

Makes the case that a major cause of problems arises due to lack of attention to requirements, planning, and early design. It is suggested that it is easier to look at hazards in a system rather than all of the possible errors with that system. A brief outline of a method is presented. The method includes preliminary hazard analysis and system hazard cross checks, both techniques being relatively simple and cheap to apply. The point is made that the format of the requirements is relatively unimportant, but that the requirements must be clear and verifiable. Indeed, for each requirement there should be a statement of how that requirement may be verified. The point is made that software is, by itself, always safe, but it is in the context of a system that software may become unsafe. This means that the hazard analysis originally undertaken must be carried through the software development.

[93] Institute of Electrical Engineers. *Health and Safety Legislation, and Consumer Legislation: Guidance for the Engineer: Professional Brief.* London, 1988.

[94] Institution of Electrical Engineers. *Programmable Electronic Systems and Safety —HSE Guidelines: Proceedings of a Colloquium*, London, 1987.

[95] Institution of Electrical Engineers. Software Requirements for High-Integrity Systems: *Proceedings of a Colloquium*, London, 1988.

[96] International Civil Aviation Authority. *The Assessment of the Integrity of Systems Containing Safety Critical Software.* International Civil Aviation Organization, October 1987.

[97] International Electrotechnical Commission. "Software for Computers in the Safety Systems of Nuclear Power Stations, IEC 880," 1986.

This standard outlines the software development techniques to be used in the development of software for the shutdown systems of nuclear power plants. The standard does not mandate particular techniques rather it states the requirements on the product and it is up to the developer to meet those requirements using whatever techniques the developer considers suitable. There are guidelines presented in an appendix that describe the effects that the techniques are expected to achieve.

[98] M.S. Jaffe and N.G. Leveson. Completeness, Robustness, and Safety in Real-time Software Requirements Specification. In *Proceedings Eleventh International Conference on Software Engineering*, pages 302-311, Pittsburgh, PA, May 1989.

Essentially the same information as found in the IEEE paper [100]. This paper lists various criteria for completeness of system requirements. The presentation of criteria is less well done than in the later paper.

[99] M.S. Jaffe and N.G. Leveson. *Completeness, Robustness, and Safety in Real-time Software Requirements Specification.* Technical Report 89-01, Information and Computer Science Department, University of California, Irvine, CA, 1989.

This report is a slightly fuller version of the ICSE paper [98]. The report adds a section on scope, defining the difference between "internal" and "external" completeness and discusses that the work is concerned solely with internal completeness.

[100] M.S. Jaffe, N.G. Leveson, M. Heimdahl, and B. Melhart. Software Requirements Analysis for Real-Time Process-Control Systems. *IEEE Transactions on Software Engineering*, March 1991.

Makes the point that prototyping suffers from the same problems as testing with respect to ensuring high degrees of confidence about the system. Similarly ex-

ecutable specifications are placed in the same category, though executable specifications and prototypes are considered to be better than formalism for determining the value of the user interface. Software correctness is considered as being a combination of subsystem correctness (the implementation of the subsystem satisfies the subsystem requirements) and system correctness (the subsystems co-operate to satisfy the system requirements). The robustness of the system depends on the completeness of the specification with respect to assumptions about the environment. The body of the paper presents a number of criteria that help find errors (or potential errors) in the requirements specification. This paper does not refer to hazard analysis, but rather to analyses that may be performed on a particular type of formal model of the system, the claim is that the formalism used is general and used to show the analyses that must be performed, and that a requirements engineer must perform analyses of these types regardless of the formalism used.

[101] F. Jahanian and A.K. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):890-904, September 1986.

[102] B.W. Johnson and J.H. Aylor. Reliability and Safety Analysis in Medical Applications of Computer Technology. In *Symposium on the Engineering of Computer-Based Medical Systems*, pages 96-100, Minneapolis, MN, June 1988.

[103] G. Jones. Fault Tree, or Not Fault Tree, That Is the Question. *Safety and Reliability*, 3(4), 1983.

[104] P.C. Jorgensen. Early Detection of Requirements Specification Errors. In *COMPASS '88 Computer Assurance*, pages 44-48, Gaithersburg, MD, July 1988.

Describes the use of extensions to Petri nets to form Petri net processes. These process representations may be used in the specification and analysis of systems involving concurrently executing cooperating processes. The paper discusses some tools that have been developed to support the use of Petri net processes. The use of the tools enables the designer to develop software safety requirements and to detect potential errors when combining processes. The paper gives examples of the use of the tools on small problems and raises doubt about the scalability of these tools to real systems.

[105] P.C. Jorgensen and W.A. Smith. Using Petri Net Theory to Analyze Software Safety Case Studies. In COMPASS '89: IEEE Fourth Annual Conference on Computer Assurance, pages 22-25, Gaithersburg, MD, 1989.

[106] F.E. Kattuah. Applicability of System Safety Methods to Software/Firmware Safety. In *Seventh International Software Safety Conference*, pages 2.6-4-1—2.6-4-18, 1985.

[107] R.A. Kirkman. Evaluation Single Point Failures for Safety & Reliability. *IEEE Transactions on Reliability*, R-28(3), August 1979.

[108] T.A. Kletz. Hazard Analysis—A Review of Criteria. *Reliability Engineering*, 3(4):325-338, 1982.

[109] B.G. Kolkhorst and A.J. Macina. Developing Error-Free Software. In *COMPASS '88 Computer Assurance*, pages 99-107, Gaithersburg, MD, July 1988.

Presents statistics generated by IBM on the development of software for the space shuttle. The figures show an impressive trend in producing software with very few defects. The implication is that the process used by IBM is applicable to other systems requiring high assurance. One reason cited for the quality was that both developers and managers were committed to achieving these levels of quality. The lessons learned were to use structured software development and test environments, to avoid using software design to mitigate early hardware or system shortcomings, and to use more software and hardware conforming to industry standards.

[110] R. Konakovsky. Safety Evaluation of Computer Hardware and Software. In *Proc. COMPSAC '78*, pages 559-564, 1978.

[111] M. La Manna. A Robust Database for Safe Real-time Systems. In *IFAC SAFECOMP* '86, pages 67-72, Sarlat, France, 1986.

[112] F.P. Lamazor. The Software Developer's Role in Assuring Nuclear Safety. In *COMPCON '84: The Small Computer (R)Evolution*, pages 365-369, Arlington, VA, 1984.

Describes global requirements for safety in nuclear weapons based on military standards. The paper then lists generic requirements for the software components of such systems. These global and generic requirements lead to the use of specific programming techniques described in the paper. These programming techniques help protect against programming errors as well as unauthorized program modifications.

[113] F.P. Lamazor. Computer System Test and Evaluation Planning, Safety Considerations. In *COMPASS '87 Computer Assurance*, pages 26-30, Washington, D.C., July 1987.

Discusses a number of issues with respect to engineering requirements and testing. The argument is that assigning appropriate levels of test criteria such as analysis, inspection, demonstration, etc., to the requirements at an early stage help clarify ambiguous or unclear requirements. The paper also argues strongly for tools to assist in tracing requirements. The improvements in the requirements analysis process should improve a system's safety.

[114] C.E. Landwehr. Software Safety is Redundance. In *COMPCON '84: The Small Computer (R)Evolution*, page 195, Arlington, VA, 1984.

The paper argues that software safety is a redundant term, that safety should be stated in the requirements and that then the issue is one of assuring that the implementation meets the requirements specification. There is a brief discussion of techniques that have been employed for the development of high assurance software for security systems. The argument is that these techniques are essentially introducing redundancy into the specification, design and implementation of the system.

[115] J.-C. Laprie, N.G. Leveson, E. Pilaud, and M. Thomas. Real-Life Safety-Critical Software Panel Position Papers. In *12th International Conference on Software Engineering*, pages 222-227, Nice, France, March 1990.

[116] Nancy G. Leveson. Software Safety: Why, What, and How. *ACM Computing Surveys*, 18:25-69, June 1986.

[117] Nancy G. Leveson. Software Safety in Embedded Computer Systems. *Communications of the ACM*, 34(2):34-46, February 1991.

An extensive article on hazard analysis, outlining the distinction between system safety and software safety. The article discusses the manner in which hazard analysis may be performed and outlines a number of the assumptions on which traditional hazard analysis is based and why these may not hold true for software.

[118] Nancy G. Leveson and P.R. Harvey. Analyzing Software Safety. *IEEE Transactions on Software Engineering*, SE-9(5):569-579, September 1983.

[119] Nancy G. Leveson and J.L. Stolzy. Safety Analysis Using Petri Nets. *IEEE Transactions on Software Engineering*, pages 386-397, March 1987.

[120] N.G. Leveson. *Software Safety: A Definition and Some Preliminary Ideas.* Technical Report 174, Dept. of Computer Science, University of California, Irvine, CA, 1981.

[121] N.G. Leveson. Design for Safe Software. In *Proc. AIAA Space Sciences Meeting*, January 1983.

[122] N.G. Leveson. Software Fault Tolerance: The Case for Forward Recovery. In *Proc. AIAA Conference on Computers in Aerospace*, Hartford, October 1983.

[123] N.G. Leveson. Verification of Safety. In *SAFECOMP '83*, Cambridge, England, 1983.

[124] N.G. Leveson. Murphy: Expecting the Worst and Preparing for It. In *COMPCON '84: The Small Computer (R)Evolution*, pages 294-300, Arlington, VA, September 1984.

Presents arguments that software safety is an important issue, that safety is not equivalent to reliability and that safety is not equivalent to bug-free. Definitions of accident, mishap, damage, hazard, and hazard severity and probability are given to define risk. Software itself is safe; risk arises when software controls systems that are inherently unsafe. Three general requirements on such software are given and there is a discussion of the conflict between safety requirements and the functional or performance requirements on the system. A development process for safety systems is outlined discussing the need for safety to be a concern throughout the development process, that the requirements must also indicate what the software shall not do, and an outline of the method of performing a hazard analysis is required. Petri-nets and fault tree analysis are described as techniques that may be used to determine the conditions that may lead to a mishap. The advantages of fault-tree analysis over verification for the software are presented. It is suggested that Petri-net analysis complements rather than competes with fault-tree analysis. Murphy, an idea for a runtime safety support environment, is presented with discussion of the possible dangers of the Murphy approach.

[125] N.G. Leveson. Software Safety in Computer-Controlled Systems. *Computer*, pages 48-55, February 1984.

[126] N.G. Leveson. Software Safety in Medical Systems. In *Seventeenth Annual Hawaii International Conference on System Sciences*, pages 13-19, 1984.

[127] N.G. Leveson. An Outline of a Program to Enhance Software Safety. In IFAC SAFECOMP '86, pages 120-135, Sarlat, France, 1986.

[128] N.G. Leveson. *Building Safe Software*. Technical Report 86-14, Department of Information and Computer Science, University of California, Irvine, February 1986.

The report outlines approaches to the development of safe software. Work is divided into two categories: modeling and analysis techniques and design techniques. For modeling and analysis, software safety requirements analysis using fault trees or timed Petri-nets are discussed. Issues of verification and validation of safety are discussed and software fault trees are proposed as an applicable approach. The discussion of assessment of safety makes the case that it may be beneficial to use resources to eliminate, minimize, or control hazards rather than expend those resources attempting to prove that a system meets a particular level of risk. Designing for safety may meet existing system safety precedence. It may be possible that the system will be intrinsically safe, or hazards may be minimized, or controlled, or finally, the hazard may be made at least visible through warnings. Two principles are discussed: the design should minimize the amount of verification required, and features added to increase safety should be evaluated in terms of the additional complexity added. Issues with respect to hazard avoidance and detection and treatment are discussed—

the aim being either to completely avoid the hazardous state or to minimize the time spent in that state. Both forward and backward error recovery fault tolerance techniques are discussed. Designs that include a guaranteed safe state and those with reduced functionality are described.

[129] N.G. Leveson. Building Safe Software. In *COMPASS '86 Computer Assurance*, pages 37-50, 1986.

[130] N.G. Leveson. *Software Safety.* SEI Curriculum Module SEI-CM-6-1.1 (Preliminary), Software Engineering Institute, July 1987.

[131] N.G. Leveson. What Is Software Safety? In COMPASS '87 Computer Assurance, pages 74-75, Washington, D.C., July 1987.

This position paper argues that building perfect software is an unrealistic goal; however, software need not be perfect to be safe. It continues by discussing that safety must be built in from the start and that it is not generally possible to add safety as an afterthought. Software safety is defined as the risk of software-related hazards when the software is executing within a particular system context. The software is not unsafe by itself, but the system context may make the operation of the software hazardous. Software safety may be increased by applying system safety techniques to the software. The paper notes that building safety-critical software with an acceptable level of risk requires changes to the development life cycle which will almost certainly prove more expensive than the development of non-safety-critical software.

[132] N.G. Leveson. Safety as a Software Quality. *IEEE Software*, pages 88-89, May 1989.

[133] N.G. Leveson. Safety-Critical Software Development. In T. Anderson, editor, Safe and Secure Computing Systems, pages 155-162. Blackwell Scientific Publications, 1989.

This paper considers the differences between developing systems and developing safety-critical systems. It is stressed that the outline of necessary development changes presented in the paper is not expected to be complete or optimal, but rather to suggest requirements on the development activity. The paper suggests three major changes to development methodologies; 1) to differentiate between failures and to treat high cost or serious failures differently, 2) to perform backward analysis using identified hazards to work back through the code or design to demonstrate that the software cannot produce the hazardous output, 3) to perform system-wide modelling and analysis. The paper then outlines the steps in a process unique to safety-critical software development. Finally, the point is made that no one technique will suffice, but that safety concerns must span the entire development activity and be considered in all aspects of building software.

[134] N.G. Leveson. Building Safe Software. In Chris Anderson, editor, Aerospace Software Engineering. AIAA, 1990.

[135] N.G. Leveson, S.S. Cha, J.C. Knight, and T.J. Shimeall. The Use of Self Checks and Voting in Software Error Detection: An Empirical Study. IEEE Transactions on Software Engineering, SE-16(4), April 1990.

A highly detailed account of a careful experiment designed to test the effectiveness of the checks in self-checking code. A number of cases used N-version approaches and yet even when a designer was deliberately attempting to use a different algorithm, the second algorithm failed in the same way as the original 60% of the time. Checks written based on the specification alone discovered 30% of the known bugs a further 45% were of known bugs were discovered by checks based on reading the code. Only 45% of the experimenters found any of the bugs at all, and there was little overlap between programmers working on the same versions. Self-checking code found bugs that had not been previously detected—the implication being that checking intermediate values is a more effective way of finding faults than comparing final values.

[136] N.G. Leveson, S.S. Cha, and T.J. Shimeall. Safety Verification of Ada Programs Using Software Fault Trees. *IEEE Software*, 8(4):48-59, July 1991.

An excellent introduction to the notion of using software fault trees for the analysis of safety-critical programs. A number of the basic concepts of hazard analysis are introduced. Software fault trees specific to Ada are presented. The role of software fault-tree analysis in system development is also discussed.

[137] N.G. Leveson and P.R. Harvey. Software Fault Tree Analysis. *The Journal of Systems and Software*, pages 173-181, 1983.

[138] N.G. Leveson and T. Shimeall. Safety Assertions for Process Control Systems. In *Proc. 13th Int. Conference on Fault Tolerant Computing*, Milan, Italy, 1983.

[139] N.G. Leveson, T.J. Shimeall, J.L. Stolzy, and J. Thomas. Design for Safe Software. In *AIAA Space Sciences Meeting*, Reno, January 1983.

[140] N.G. Leveson and J.L. Stolzy. Safety Analysis of Ada Programs Using Fault Trees. *IEEE Transactions on Reliability*, R-32(5):479-484, December 1983.

[141] N.G. Leveson and C.S. Turner. An Investigation of the Therac-25 Accidents. Technical Report (92-108), Information and Computer Science Dept., University of California, Irvine, CA.1992.

This report is an excellent discussion of the history of the Therac-25 linear accelerator. The discussion of the accidents includes details of the accident in addition to comments on the reactions of the various parties involved. The general recommendations are applicable to a wider audience than developers of medical systems.

[142] S. Levine. Probabilistic Risk Assessment: Identifying the Real Risks of Nuclear Power. *Technical Review*, pages 41-44, Feb-Mar 1984.

[143] G. Liedstrom. The Human Side of Medical Device Safety. *International Medical Device & Diagnostic Industry*, pages 6-7, May-Jun 1990.

[144] O.C. Lindsey. Hazard Analysis for Software Systems. In *Proc. Third International System Safety Conference*, 1977.

[145] Chemical Industries Association Ltd. A Guide to Hazard and Operability Studies.

[146] H.O. Lubbes. Computer Safety Acquisition Model. In *Sev9nth Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1241-1247, Chicago, Ill., 1985.

[147] D. Luckham. *Programming with Specifications.* Springer-Verlag, 1990.

[148] K.L. MacMillan. Software Safety Analysis. In *Seventh Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1222-1229, Chicago, Ill., 1985.

[149] D.P. Mannering and B. Cohen. The Rigorous Specification and Verification of the Safety Aspects of a Real-Time System. In *Fifth Annual Conference on Computer Assurance*, pages 68-85, Gaithersburg, MD, June 1990.

[150] E.J. Marchant et al. Safety Analysis of Computerized Systems. *Hazard Prevention: Journal of the System Safety Society*, Mar-Apr 1984.

[151] R.L. McCarthy. Present and Future Safety Challenges of Computer Control. In *COMPASS '88 Computer Assurance*, pages 1-7, Gaithersburg, MD, July 1988.

A general, introductory paper presenting definitions of the term accident and risk. The paper discusses risks to the public of non-computer related systems (including living) and suggests that due to the advances in medicine through computer control people are generally better off. However, the risks associated with computer control of safety systems are still important due to the increased dependency on such systems. The paper presents many statistics as an indication of where risks arise. It suggests that proper design of computer controlled systems will reduce errors due to operational mode failure. The paper also discusses some social issues such as legal liability with respect to computer controlled systems affecting safety.

[152] J.A. McDermid. The Role for Formal Methods in Software Development. *Journal of Information Technology*, 2(3):124-134, September 1987.

[153] J.A. McDermid. *Principles of an Assurance Algebra for Dependable Software.* Technical Report, University of York, Department of Computer Science, September 1989.

[154] J.A. McDermid. Towards Assurance Measures for High Integrity Software. In *Reliability 89.* Institute of Quality Assurance, 1989.

[155] G. McDonald. Systems Software Safety and The Life Cycle. In *Proceedings, NSIA Second Conference on Software Quality and Productivity*, pages 505-592, March 1986.

[156] G.W. McDonald. Why There Is a Need for a Software Safety Program. In *Annual Reliability and Maintainability Symposium*, pages 30-34, Atlanta, Ga., January 1989.

[157] J.W. McIntee. *Fault Tree Technique as Applied to Software (SOFT TREE).* BMO/AWS, Norton Air Force Base, CA 92409.

[158] Medical Device Industry Computer Software Committee. *Reviewer Guidance for Computer-Controlled Devices (Draft)*. Medical Device Industry Computer Software Committee, January 1989.

[159] B.E. Melhart. *Specification and Analysis of the Requirements for Embedded Software with an External Interaction Model.* Doctoral dissertation, Univ. of California, Irvine, CA, July 1990.

[160] S. Migues. The Need for Rigorous Informal Verification of Specification-to-Code Correspondence. In *Compass '87 Computer Assurance*, pages 13-25, Washington, D.C., July 1987.

The paper bases its conclusions on a number of examples of security systems. The fundamental argument is that neither traditional nor formal methods are currently applicable to the problem of demonstrating that an implementation satisfies its specification. Instead, a rigorous approach is suggested, specifically, using assertions based on the specification embedded in the code as a check on the implementation.

[161] H.D. Mills. Engineering Discipline for Software Procurement. In *COMPASS '87 Computer Assurance*, pages 1-5, Washington, D.C., July 1987.

Discusses the fact that the development of software is a new human activity and that software is complex. The distinction between software engineering and programming is drawn with considerable emphasis on a formal approach to software development as a key distinguishing factor. Although this paper does not directly relate to development of software for safety-critical systems, it does outline the author's views of software engineering and a development approach that is applicable to safety-critical software.

[162] Ministry of Defence. *Defence Standard 00-31: Development of Safety Critical Software for Airborne Systems.* Ministry of Defence, Great Britain, 1987.

The standard applies to all MOD avionics systems that contain software and have flight safety implications. The standard recognizes that is also applicable to other non-avionics software systems. The standard is a variant of RTCA/DO-178A [188], particularizing the RTCA standard for the United Kingdom. It does add that the MOD project director has the right to audit the documentation pro-

duced by the contractor or the management systems used by the contractor to determine compliance with the standard.

[163] Ministry of Defence. *Defence Standard 00-55: The Procurement of Safety Critical Software in Defence Equipment.* Ministry of Defence, Great Britain, April 1991.

[164] Ministry of Defence. *Defence Standard 00-56: Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.* Ministry of Defence, Great Britain, April 1991.

[165] C. Morgan. *Programming from Specifications.* Prentice Hall International, 1990.

[166] National Association of Lift Makers. *Programmable Electronic Systems in Safety-Related Applications*, August 1988.

[167] NATO AC/310 Ad Hoc Working Group on Munition Related Safety Critical Computing Systems. *STANAG 4404 NATO Standardization Agreement: Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems (Draft)*, March 1990.

[168] P. Neilan, editor. The Assessment of Safety-Related Systems Containing Software. *Proceeding of the CSR Conference on Certification*, 1988.

[169] P.G. Neumann. On Hierarchical Design of Computer Systems for Critical Applications. *IEEE Transactions on Software Engineering*, SE-12(9):905-920, September 1986.

Presents reasons why a safety-critical system should be secure and fault-tolerant as well as safe. The paper discusses a number of classes of criticality, of which safety is one. The paper continues with a discussion of hierarchical designs for various systems and how the different layers provide different levels of trust in the system. A skeletal design hierarchy for all classes of criticality is developed and presented and discussion of verification techniques indicates that the most trusted components of the system must be small due to limitations in software verification techniques. Neumann does not believe that all safety critical code can always be confined to a kernel, instead he believes that a trusted computer base designed hierarchically is the best that can be done. Careful structuring of the design will help confine bad effects.

[170] P.G. Neumann. The Computer Related Risk of the Year: Computer Abuse. In *COMPASS '88 Computer Assurance*, pages 8-12, Washington, D.C., July 1987.

The paper concentrates on security issues, describing the three gaps that Neumann sees that lead to penetration of systems. The argument is made that if a system is not secure, then it cannot be safe as it cannot be trusted to do what is expected. The gaps are technological (we cannot be sure that the mecha-

nisms meet the policy); socio-technological (we cannot ensure that the computer policy enforces the desired social policies); and social (we do not always get reasonable human behavior even though we generally assume reasonable behavior when designing systems).

[171] P.G. Neumann. What Is Software Safety? In *COMPASS '87 Computer Assurance*, page 76, Washington, D.C., July 1987.

This position paper argues that software safety can only be discussed in the context of system safety, and that no non-trivial system can ever be guaranteed to be completely safe in all circumstances. Software safety must be defined very precisely for each application, and that generic definitions seem inadequate in the absence of the context of the system. Thus, although it is desirable to reuse specifications, models, and programs, this may not be easy to accomplish. Finally, the point is made that safety cannot be considered in isolation, that it is just one critical requirement and refers to an earlier paper [169] which discusses other critical requirements.

[172] Nuclear Regulatory Commission. *CR-4780: Procedures in Safety and Reliability Studies*, (2 vols), January 1983.

[173] S.R. Nunns, D.A. Mills, and G.C. Tuff. Programmable Electronic Systems Safety: Standards and Principles—An Industrial Viewpoint. In *IFAC SAFECOMP '86*, pages 17-20, Sarlat, France, 1986.

[174] G. O'Neill and B.A. Wichmann. *A Contribution to the Debate on Safety Critical Software.* NPL Report DITC 126/188, National Physical Laboratory, Great Britain, September 1988.

[175] M. Ould. Safe Software: The State of the Art. *Information Technology and Public Policy*, 6(3):215-218, Summer 1988.

[176] G. Page, F.E. McGarry, and D.N. Card. A Practical Experience with Independent Verification and Validation. In *COMPSAC '84*, pages 453-457, November 1984.

[177] D.L. Parnas, G.J.K. Asmis, and J. Madey. *Assessment of Safety-Critical Software.* Technical Report 90-295, Queens University, Kingston, Ontario, Canada, Ontario, Canada, December 1990.

A good example of the need for formal requirements for safety-critical systems. The authors are concerned with a specific style of specification and development. The description of the process at Darlington is interesting and valuable. It should be tempered with the knowledge that many of the developers were reported to be unhappy with the total process.

[178] D.L. Parnas, A.J. van Schouwen, and S.P. Kwan. *Evaluation Standards for Safety Critical Software.* Technical Report 88-220, Department of Computing

and Information Science, Queen's University at Kingston, Kingston, Ontario, Canada, 1988.

Discusses reasons why software is used in place of hardware for safety-critical systems, even though software tends to have less predictability than hardware. The discussion centers on the need for system documentation that is reviewable by domain experts rather than software experts. The particular approach used is the A7E style. An interesting point is raised that even when non-communicating programmers develop pieces of code that common errors often arise—this seems to point to ambiguities in requirements, though this conclusion is not drawn here. The examples of appropriate documentation appear to be at the module design level rather than the system level.

[179] D.L. Parnas, A.J. van Schouwen, and S.P. Kwan. Evaluation of Safety Critical Software. *Communications of the ACM*, 33(6):636-648, June 1990.

Essentially the same as the 1988 technical report.

[180] C. Perrow. Normal Accidents: Living with High Risk Technologies. Basic Books, 1984.

The book discusses a number of accidents that have occurred in various domains, providing detailed accounts of the actions that took place leading up to the accident, the details of the accident, and lessons we may learn from the accident. The book makes the point that in the design of systems, we need to examine the history of failures in those systems to avoid similar failures. However, the argument continues that history is not enough as the addition of safety devices to avoid past failures will introduce new ways for a system to fail. Designs have become so complicated that we cannot anticipate all of the interactions between the components of the system or between the system and its environment. Failures are a particular type of interaction and we cannot anticipate all failures either. In many cases, safety devices are added after the fact to handle anticipated failures, however, it is often the case that these safety devices are deceived, defeated, or avoided by unanticipated interactions in the systems. The book considers a number of areas of human activity and measures these using metrics based on the complexity of the systems and the how tightly coupled the systems are. The argument is that in systems that are complex and tightly coupled we find the greatest potential for unsafe behavior.

[181] D. Petersen. *Techniques of Safety Management.* McGraw-Hill, New York, 1971.

[182] H. Petroski. Successful Design as Failure Analysis. In *COMPASS '87 Computer Assurance*, pages 46-48, Washington, D.C., July 1987.

The subject of this paper is a history of designs from civil engineering. The purpose of the paper is to indicate that while in the process of design, the designer should be aware of prior designs within the chosen field that have failed and the reasons for the failures. This may help the designer from repeating the earlier mistakes and will certainly help the designer in considering various failure modes that may affect their design. Although the paper does not discuss soft-

ware development at all, the reader is constantly aware of the parallels in the processes. The process of design is full of pitfalls, some of which may be avoided by remembering history.

[183] B.H. Peyton and D.C. Hess. Software Sneak Analysis. In *Seventh Annual Conference of the IEEE Engineering in Medicine and Biology Society*, pages 193-196, Chicago, Ill., 1985.

[184] E. Pilaud. Some Experiences of Critical Software Development. In *12th International Conference on Software Engineering*, pages 225-226, Nice, France, March 1990.

[185] J. Pill. The Delphi Method: Substance, Context, A Critique and an Annotated Bibliography. *Socio-Economic Planning Sciences*, 5:61, 1971.

[186] P.R.H. Place, W.G. Wood, and M. Tudball. *Survey of Formal Specification Techniques for Reactive Systems.* Technical Report CMU/SEI-90-TR-5, ESD-TR-90- 206, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, May 1990.

A technical report comparing three formal specification techniques. The report demonstrates that it is possible to represent the customers natural language requirements using a formal notation in such a way that the formal specification may be validated.

[187] W.J. Quirk. Engineering Software Safety. In *IFAC SAFECOMP '86*, pages 143-147, Sarlat, France, 1986.

[188] Radio Technical Commission for Aeronautics, Washington, D.C. *Software Considerations in Airborne Systems and Equipment Certification*, do-178a edition, 1985.

[189] C.V. Ramamoorthy, G.S. Ho, and Y.W. Han. Fault Tree Analysis of Computer Systems. In Proc. *National Computer Conference*, pages 13-17, 1977.

[190] D.J. Reifer. Software Failure Modes and Effects Analysis. IEEE Transactions on Reliability, R-28(3):247-249, August 1979.

[191] W.A. Reupke, E. Srinivasan, P. Rigterink, and D.N. Card. The Need for a Rigorous Development and Testing Methodology for Medical Software. In *Symposium on the Engineering of Computer-Based Medical Systems*, pages 15-20, Minneapolis, MN, June 1988.

[192] C.L. Rhoades. Software Hazard Identification and Control. In *Seventh International Systems Safety Conference*, pages 2.6-1-1—2.6-1-13, 1985.

[193] W.P. Rodgers. *Introduction to System Safety Engineering.* Wiley, New York: 1971.

[194] H.E. Roland and B. Moriarty. New York· *System Safety Engineering and Management.* New York: Wiley, 1983.

[195] C.W. Rose. The Contribution of Operating Systems to Reliability and Safety in Real-Time Systems. In *SAFECOMP '82*, 1982.

[196] B. Runge. Quantitative Assessment of Safe and Reliable Software. In *IFAC SAFECOMP '86*, pages 7-11, Sarlat, France, 1986.

[197] J.M. Rushby. Kernels for Safety? In T. Anderson, editor, *Safe and Secure Computing Systems*, pages 210-220, Glasgow, Scotland, October 1986.

This paper concentrates on security kernels. It should be noted that this paper uses the term "kernel" to mean a mechanism for policy enforcement rather than a structuring concept. However, there is an initial discussion of what a kernel can and cannot do, it is made clear that a kemel cannot enforce good behavior but that it can prevent bad behavior. A system structure using a kemel is only appropriate if the system level properties can be defined as kernel level functions. The paper discusses mathematical descriptions of properties and argues that positive properties can be expressed in first order logic but that negative (safety) properties need to use second order logic. The paper then discusses the notion of a separation kemel which maintains barriers between different domains within the system. Then properties such as "this domain may not influence that domain" can be enforced.

[198] A.P. Sage. Methodologies for Risk and Hazard Assessment: A Survey and Status Report. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10(8):425-446, August 1980.

[199] D. Santel, C. Trautmann, and W. Liu. Formal Safety Analysis and the Software Engineering Process in the Pacemaker Industry. In *COMPASS '88 Computer Assurance*, pages 129-131, Gaithersburg, MD, July 1988.

Using a pacemaker as an example, the paper discusses that due to the increased complexity of systems, development engineers necessarily come from a number of different specialized fields and that none of them may have the breadth of background to assess system hazards. The paper makes the point that safety requirements are couched in terms of what the system will not do, rather than what the system will do (as for functional requirements). Due to the wide variation in systems, there is little historical data from which probabilities of a mishap occurring may be drawn. Instead of using quantitative values, qualitative probability is used. Qualitative probably is then applied to the hazards as identified using hardware hazard analysis techniques to develop software safety requirements. Requirements are then traced through all levels of development using a requirements traceability matrix. The paper concludes by stating that they do not expect that any technique will not eliminate every hazard, but instead they will minimize the number of hazards and offer the developers and end-users confidence in the final product.

[200] D. Santel, C. Trautmann, and W. Liu. The Integration of a Formal Safety Analysis into the Software Engineering Process: An Example from the Pacemaker Industry. In *Symposium on the Engineering of Computer-Based Medical Systems*, pages 152-154, Minneapolis, MN, June 1988.

[201] D.J. Schultz. *Software Safety Engineering.* Technical Report, Computer Sciences Corp., 1987.

[202] C.T. Sennett. *High Integrity Software.* Pitman, 1989.

This book deals with techniques used that may be used to develop high integrity software. The book covers the use of formal methods for specification and design of the software and includes a report on practical experience of formal verification. Various approaches for achieving fault tolerance are discussed. There are chapters relating to implementation languages and approaches that may be used to analyze implementations. The book concludes with chapters on achieving high assurance.

[203] P.V. Shebalin and S.H. Son. An Approach to Software Safety Analysis in a Distributed Real-Time System. In *COMPASS '88 Computer Assurance*, pages 29-43, Gaithersburg, MD, July 1988.

The paper deals with safety issues in a distributed system where the processes communicate by means of message passing. The system is modelled by a number of components, safety critical devices, connectivity relations, messages, and rules describing component behavior. Component message fault analysis, a six-step analysis of the distributed software is detailed. The method uses forward and backward flow graphs to model the component software, these appear to be similar to software fault trees. The paper then uses the method on a simple example. This paper presents a clear process for the analysis of system requirements and the code that implements those requirements with an approach to the demonstration of the safety of the code according to the system safety requirements.

[204] V. Shen. Safety as a Software Quality. *IEEE Software*, pages 88-89, May 1989.

[205] E.H. Sibley, J.B. Michael, and R.L. Wexelblat. Policy Management, Economics, and Risk. In *Second International Conference on Economics and Artificial Intelligence*, Paris, France, July 1990.

[206] P. Slovic. Informing and Educating the Public About Risk. *Risk Analysis*, 6(4):403-415, 1986.

[207] A.J. Somerville. Fail-Safe Design of Closed Loop Systems. In *Symposium on the Engineering of Computer-Based Medical Systems*, pages 23-27, Minneapolis, MN, June 1988.

[208] C. Starr. Risk Management, Assessment and Acceptability. Risk Analysis, 5(2):57-102, 1985.

[209] J.R. Taylor. *Logical Validation of Safety Control System Specifications Against Plant Models.* Technical Report RISO-M-2292, Riso National Laboratory, Roskilde, Denmark, 1981.

[210] J.R. Taylor. *Fault Tree and Cause Consequence Analysis for Control Software Validation.* Technical Report RISO-M-2326, Riso National Laboratory, Roskilde, Denmark, 1982.

[211] N. Theuretzbacher. Using AI Methods to Improve Software Safety. In *IFAC SAFECOMP '86*, pages 99-105, Sarlat, France, 1986.

[212] M. Thomas. Assessing Failure Probabilities in Safety-Critical Systems Containing Software. In *12th International Conference on Software Engineering*, page 227, Nice, France, March 1990.

[213] N.C. Thomas and E.A. Straker. Application of Verification and Validation to Safety Parameter Display Systems. In *American Nuclear Society International Topical Meeting on Computer Applications for Nuclear Power Plant Operation and Control*, pages 274-282, Richland, WA, September 1985.

[214] F.A. Tuma. Verifying Software System Safety. In *COMPCON '84: The Small Computer (R)Evolution*, pages 370-375, Arlington, VA, 1984.

Starts out by asserting that safety analysis should be performed as early as possible and by discussing use of a prioritized list of critical system functions to determine which parts of the system get analyzed. Sneak Software Analysis is introduced as a way of verifying the operational code. Essentially, a network tree is built up from the code, so that the network describes the program control flow. The analyst examines the network tree for particular problems, such as logic being bypassed. The paper claims that it is relatively easy to compare the actual function (through the network tree) with the requirements and design. The paper continues with a description of a traceability tool which essentially is a computer manipulated record of the connectivity between system requirements, software requirements, functional designs, implementation code, and test procedures. The paper continues discussing tools for analyzing changes to the software and tools for analyzing interfaces between software, hardware, and firmware.

[215] W.E. Vesely, F.F. Goldberg, N.H. Roberts, and D.F. Haasl. *Fault-Tree Handbook*, Reg. 0492. US Nuclear Regulatory Comm., Washington, D.C., January 1981.

[216] D.R. Wallace and J.C. Cherniavsky. *Guide to Software Acceptance.* NIST Special Publication 500-180, U.S. Department of Commerce, April 1990.

[217] P. Wetterlind and W.M. Lively. Ensuring Software Safety in Robot Control. In *1987 Fall Joint Computer Conference*, pages 34-37, Dallas, TX, October 1987.

[218] M.J. Whitelaw. Process Computer Replacement and Implementation of SPDS Requirements at Millstone 2 Nuclear Power Plant. *IEEE Transactions on Nuclear Science*, 35(1):919-923, February 1988.

[219] N.P. Wilburn. Software Verification for Nuclear Industry. In *American Nuclear Society International Meeting on Computer Applications for Nuclear Power Plant Operation and Control*, pages 229-235, Richland, WA, September 1985.

[220] A.G. Zellweger. FAA Perspective on Software Safety and Security. In *COMPCON '84: The Small Computer (R)Evolution*, pages 200-201, Arlington, VA, September 1984.

Describes the state of an air traffic control system under development. The paper argues that experience has shown that the system can operate safely, though inefficiently, with a small subset of the function. Thus, in the case of massive system failure, the system reverts to an emergency mode where only these essential functions are provided.

[221] R. Zerwekh. Problem Programs: Negligence and the Computing Profession. In *COMPASS '88 Computer Assurance*, Gaithersburg, MD, July 1988.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-93-TR-5

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-93-182

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
ESC/AVS
Hanscom Air Force Base, MA 01731

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/AVS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Safety-Critical Software: Status Report and Annotated Bibliography

12. PERSONAL AUTHOR(S)
Patrick R. H. Place, Kyo C. Kang

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM TO

14. DATE OF REPORT (year, month, day)
June 1993

15. PAGE COUNT
78

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
safety-critical software, requirements engineering, hazard identification

19. ABSTRACT (continue on reverse if necessary and identify by block number)

Many systems are deemed safety-critical and these systems are increasingly dependent on software. Much has been written in the literature with respect to system and software safety. This report summarizes some of that literature and outlines the development of safety-critical software. Techniques for hazard identification and analysis are discussed. Further, techniques for the development of safety-critical software are mentioned. A partly annotated bibliography of literature concludes the report.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT □ DTIC USERS ☒

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/AVS (SEI)

EDITION of 1 JAN 73 IS OBSOLETE

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS

ABSTRACT — continued from page one, block 19